Pascal Djiknavorian[1], Dominic Grenier[2]

[1,2]Laboratoire de Radiocommunications et de Traitement du Signal, Departement de Genie Electrique et de Genie Informatique, Faculte des Sciences et de Genie, Universite Laval, Quebec, Canada, G1K 7P4.

Reducing DSmT hybrid rule complexity through optimization of the calculation algorithm

Published in:
Florentin Smarandache & Jean Dezert (Editors)
Advances and Applications of DSmT for Information Fusion
(Collected works), Vol. II
American Research Press, Rehoboth, 2006
ISBN: 1-59973-000-6
Chapter XV, pp. 365 - 430

Abstract: *The work covered here had for objective to write a Matlab^{TM} program able to execute efficiently the DSmT hybrid rule of combination. As we know the DSmT hybrid rule of combination is highly complex to execute and requires high amounts of resources. We have introduced a novel way of understanding and treating the rule of combination and thus were able to develop a Matlab^{TM} program that would avoid the high level of complexity and resources needs.*

15.1 Introduction

The purpose of DSmT [3] was to introduce a theory that would allow to correctly fuse data, even in presence of conflicts between sources of evidence or in presence of constraints. However, as we know, the DSmT hybrid rule of combination is very complex to compute and to use in data fusion compared to other rules of combination [4]. We will show in the following sections, that there's a way to avoid the high level of complexity of DSmT hybrid rule of combination permitting to program it into Matlab$^{\text{TM}}$. An interesting fact to know is that the code developed and presented in this chapter is the first one known to the authors to be complete and functional. A partial code, useful for the calculation of the DSmT hybrid rule of combination, is presented in [3]. However, its function is to calculate complete hyper-power sets, and its execution took us over a day for $|\Theta| = 6$. This has made it impossible to have a basis for comparison of efficiency for our code, which is able to execute a complete DSmH combination in a very short period of time. We will begin by a brief review of the theory used in the subsequent sections, where there will be presented a few definitions followed by a review of Dempster-Shafer theory and its problem with mass redistribution. We will then look at Dezert-Smarandache theory and its complexity. It is followed by a section presenting the methodology used to avoid the complexity of DSmT hybrid rule of combination. We will conclude with a short performance analysis and with the developed Matlab$^{\text{TM}}$ code in appendix.

15.2 Theories

15.2.1 Definitions

A minimum of knowledge is required to understand DSmT, we'll thus begin with a short review of important concepts.

- *Frame of discernment* (Θ) : $\Theta = \{\theta_1, \theta_2, \ldots \theta_n\}$. It's the set including every possible object θ_i.

- *Power set* $\left(2^\Theta\right)$: represents the set of all possible sets using the objects of the frame of discernment Θ. It includes the empty set and excludes intersections. The power set is closed under union. With the frame of discernment defined above, we get the power set

$$2^\Theta = \{\emptyset, \{\theta_1\}, \{\theta_2\}, \ldots \{\theta_n\}, \{\theta_1, \theta_2\}, \ldots \{\theta_1, \theta_2, \ldots \theta_n\}, \ldots, \Theta\}.$$

- *Hyper-power set* $\left(D^\Theta\right)$: represents the set of all possible sets using the objects of the frame of discernment Θ. The hyper-power sets are closed under union and intersection and includes the empty set. With the frame of discernment $\Theta = \{\theta_1, \theta_2\}$, we get the hyper-power set $D^\Theta = \{\emptyset, \{\theta_1\}, \{\theta_2\}, \{\theta_1 \cap \theta_2\}, \{\theta_1 \cup \theta_2\}\}$.

- *Belief* $(\mathrm{Bel}(A))$: is an evaluation of the minimal level of certainty, or trust, that a set can have.

- *Plausibility* $(\mathrm{Pl}(A))$: is an evaluation of the maximal level of certainty, or trust, that a set can have.

- *Constraint* : a set considered impossible to obtain.

- *Basic belief assignment* (bba) : $m : 2^\Theta \to [0,1]$, so the mass given to a set $A \subseteq \Theta$ follows $m(A) \in [0,1]$.

- *Core of* Θ $(\mathcal{K})$: The set of all focal elements of Θ, where a focal element is a subset A of Θ such that $m(A) > 0$.

15.2.2 Dempster-Shafer Theory

The DST rule of combination is a conjunctive normalized rule working on the power set as described previously. It combines information with intersections, meaning that it works only with the bba's intersections. The theory also makes the hypothesis that the sources of evidence are mathematically independent. The i^{th} bba's source of evidence is denoted m_i. Equation (15.1) describes the DST rule of combination where K is the conflict. The conflict in DST is defined as in equation (15.2).

$$(m_1 \oplus m_2)(C) = \frac{1}{1-K} \sum_{A \cap B = C} m_1(A)\, m_2(B) \qquad \forall C \subseteq \Theta \tag{15.1}$$

$$K = \sum_{\substack{A, B \subseteq \Theta \\ A \cap B = \emptyset}} m_1(A)\, m_2(B) \tag{15.2}$$

15.2.2.1 DST combination example

Let's consider the case where we have an air traffic surveillance officer in charge of monitoring readings from two radars. The radars constitute our two sources of evidence. In this case, both radars display a target with the level of confidence (bba) of its probable identity. Radar 1 shows that it would be an F-16 aircraft (θ_1) with $m_1(\theta_1) = 0.50$, an F-18 aircraft (θ_2) with $m_1(\theta_2) = 0.10$, one of both with $m_1(\theta_1 \cup \theta_2) = 0.30$, or it might be another airplane (θ_3) with $m_1(\theta_3) = 0.10$. Collected data from radars 1 and 2 are shown in table 15.1. We can easily see from that table that the frame of discernment $\Theta = \{\theta_1, \theta_2, \theta_3\}$ is sufficient to describe this case.

The evident contradiction between the sources causes a conflict to be resolved before interpreting the results. Considering the fact that the DST doesn't admit intersections, we'll have to discard some possible sets. Also, the air traffic surveillance officer got intelligence information recommending exclusion of the case $\{\theta_3\}$, creating a constraint on $\{\theta_3\}$. Table 15.2 represents the first step of the calculation before the redistribution of the conflicting mass.

(2^Θ)	$m_1(A)$	$m_2(A)$
$\{\theta_1\}$	0.5	0.1
$\{\theta_2\}$	0.1	0.6
$\{\theta_3\}$	0.1	0.2
$\{\theta_1, \theta_2\}$	0.3	0.1

Table 15.1: Events from two sources of evidence to combine

As we can see in table 15.2, the total mass of conflict is $\sum m(\emptyset) = 0.59$. So among all the possible sets, 0.59 of the mass is given to $\emptyset$. This would make it the most probable set. Using equation (15.1) the conflict is redistributed proportionally among focal elements. Results are given in tables 15.3 and 15.4. Finally, we can see that the most probable target identity is an F-18 aircraft.

The problem, which was predictable by analytical analysis of equation (15.1), occurs when conflict (K) get closer to 1. As K grows closer to 1, the DST rule of combination tends to give incoherent results.

15.2.3 Dezert-Smarandache Theory

Instead of the power set, used in DST, the DSmT uses the hyper-power set. DSmT is thus able to work with intersections. They also differ by their rules of combination. DSmT developed

	$m_1(\theta_1)$ 0.5	$m_1(\theta_2)$ 0.1	$m_1(\theta_3)$ 0.1	$m_1(\theta_1 \cup \theta_2)$ 0.3
$m_2(\theta_1)$ 0.1	θ_1 0.05	$\theta_1 \cap \theta_2 = \emptyset$ 0.01	$\theta_1 \cap \theta_3 = \emptyset$ 0.01	θ_1 0.03
$m_2(\theta_2)$ 0.6	$\theta_1 \cap \theta_2 = \emptyset$ 0.30	θ_2 0.06	$\theta_2 \cap \theta_3 = \emptyset$ 0.06	θ_2 0.18
$m_2(\theta_3)$ 0.2	$\theta_1 \cap \theta_3 = \emptyset$ 0.10	$\theta_2 \cap \theta_3 = \emptyset$ 0.02	$\theta_3 = \emptyset$ 0.02	$(\theta_1 \cup \theta_2) \cap \theta_3 = \emptyset$ 0.06
$m_2(\theta_1 \cup \theta_2)$ 0.1	θ_1 0.05	θ_2 0.01	$(\theta_1 \cup \theta_2) \cap \theta_3 = \emptyset$ 0.01	$\theta_1 \cup \theta_2$ 0.03

Table 15.2: Results from disjunctive combination of information from table 15.1 before mass redistribution

	$m_1(\theta_1)$ 0.5	$m_1(\theta_2)$ 0.1	$m_1(\theta_3)$ 0.1	$m_1(\theta_1 \cup \theta_2)$ 0.3
$m_2(\theta_1)$ 0.1	θ_1 $\frac{0.05}{1-0.59}$	$\emptyset$ 0	$\emptyset$ 0	θ_1 $\frac{0.03}{1-0.59}$
$m_2(\theta_2)$ 0.6	$\emptyset$ 0	θ_2 $\frac{0.06}{1-0.59}$	$\emptyset$ 0	θ_2 $\frac{0.18}{1-0.59}$
$m_2(\theta_3)$ 0.2	$\emptyset$ 0	$\emptyset$ 0	$\emptyset$ 0	$\emptyset$ 0
$m_2(\theta_1 \cup \theta_2)$ 0.1	θ_1 $\frac{0.05}{1-0.59}$	θ_2 $\frac{0.01}{1-0.59}$	$\emptyset$ 0	$\theta_1 \cup \theta_2$ $\frac{0.03}{1-0.59}$

Table 15.3: Results from disjunctive combination of information from table 15.1 with mass redistribution

in [3], possesses two[1] rules of combination which are able to work around the conflicted mass redistribution problem:

- Classic DSm rule of combination (DSmC), which is based on the free model $M^f(\Theta)$

$$m(C) = \sum_{A \cap B = C} m_1(A)\, m_2(B) \qquad A, B \in D^\Theta, \forall C \in D^\Theta \tag{15.3}$$

- Hybrid DSm rule of combination (DSmH), which is able to work with many types of constraints

[1]Actually more fusion rules based on Proportional Conflict Redistributions (PCR) have been presented in Part 1 of this book. The implementation of these new PCR rules will be presented and discussed in a forthcoming publication.

$m_{1\oplus 2}(\emptyset)$	0.000
$m_{1\oplus 2}(\theta_1)$	0.317
$m_{1\oplus 2}(\theta_2)$	0.610
$m_{1\oplus 2}(\theta_1 \cup \theta_2)$	0.073

Table 15.4: Final results for the example of DST combination

$$m_{M(\Theta)}(A) = \phi(A)\left[S_1(A) + S_2(A) + S_3(A)\right] \tag{15.4}$$

$$S_1(A) = \sum_{X_1 \cap X_2 = A} m_1(X_1)\, m_2(X_2) \qquad \forall X_1, X_2 \in D^{\Theta} \tag{15.5}$$

$$S_2(A) = \sum_{[(u(X_1)\cup u(X_2))=A]\vee[((u(X_1)\cup u(X_2))\in\emptyset)\wedge(A=I_t)]} m_1(X_1)\, m_2(X_2) \qquad \forall X_1, X_2 \in \emptyset \tag{15.6}$$

$$S_3(A) = \sum_{X_1 \cup X_2 = A} m_1(X_1)\, m_2(X_2) \qquad \forall X_1, X_2 \in D^{\Theta} \quad \text{and} \quad X_1 \cap X_2 \in \emptyset \tag{15.7}$$

Note that $\phi(A)$ in equation (15.4) is a binary function resulting in 0 for empty or impossible sets and in 1 for focal elements. In equation (15.6), $u(X)$ represents the union of all objects of set X. Careful analysis of equation (15.7) tells us that it's the union of all objects of sets X_1 and X_2, when it is not empty. Finally, also from equation (15.6), I_t represents the total ignorance, or the union of all objects part of the frame of discernment. Further information on how to understand and proceed in the calculation of DSmH is available in subsequent sections.

15.2.3.1 DSmC combination example

This example cannot be resolved by DST because of highly conflicting sources of evidence (K tends toward 1). Sources' information shown in table 15.5 gives us, with DSmC, the results displayed in table 15.6. As we can see, no mass is associated to an empty set since DSmC is based on free DSm model which does not allow integrity constraints (by definition). Final results for the present example, given by table 15.7, tell us that the most probable identity of the target to identify is an hybrid of objects of type θ_1 and θ_2.

15.3 How to avoid the complexity

15.3.1 Simpler way to view the DSmT hybrid rule of combination

First of all, one simple thing to do in order to keep the use of resources at low levels is to keep only the useful data. For example, table 15.6 shouldn't be entered as is in a program but reduced to the equivalent table 15.8. This way, the only allocated space to execute the calculation is the data space we actually need.

(D^Θ)	$m_1(A)$	$m_2(A)$
$\{\theta_1\}$	0.8	0.0
$\{\theta_2\}$	0.0	0.9
$\{\theta_3\}$	0.2	0.1
$\{\theta_1, \theta_2\}$	0.0	0.0

Table 15.5: Events from two sources of evidence to combine

	$m_1(\theta_1)$ 0.8	$m_1(\theta_2)$ 0.0	$m_1(\theta_3)$ 0.2	$m_1(\theta_1 \cup \theta_2)$ 0.0
$m_2(\theta_1)$ 0.0	θ_1 0.00	$\theta_1 \cap \theta_2$ 0.00	$\theta_1 \cap \theta_3$ 0.00	θ_1 0.00
$m_2(\theta_2)$ 0.9	$\theta_1 \cap \theta_2$ 0.72	θ_2 0.00	$\theta_1 \cap \theta_3$ 0.18	θ_2 0.00
$m_2(\theta_3)$ 0.1	$\theta_1 \cap \theta_3$ 0.08	$\theta_2 \cap \theta_3$ 0.00	θ_3 0.02	$(\theta_1 \cup \theta_2) \cap \theta_3$ 0.00
$m_2(\theta_1 \cup \theta_2)$ 0.0	θ_1 0.00	θ_2 0.00	$(\theta_1 \cup \theta_2) \cap \theta_3$ 0.00	$\theta_1 \cup \theta_2$ 0.00

Table 15.6: Results from DSmC rule of combination with table 15.1 information

(θ_1)	0.00	$(\theta_1 \cap \theta_2)$	0.72	$(\theta_1 \cup \theta_2)$	0.00
(θ_2)	0.00	$(\theta_1 \cap \theta_3)$	0.08	$(\theta_1 \cup \theta_3)$	0.00
(θ_3)	0.02	$(\theta_2 \cap \theta_3)$	0.18	$(\theta_2 \cup \theta_3)$	0.00
$(\emptyset)$	0.00	$(\theta_1 \cap \theta_2) \cup \theta_3$	0.00	$(\theta_1 \cup \theta_2) \cap \theta_3$	0.00
$(\theta_1 \cup \theta_2 \cup \theta_3)$	0.00	$(\theta_1 \cap \theta_3) \cup \theta_2$	0.00	$(\theta_1 \cup \theta_3) \cap \theta_2$	0.00
$(\theta_1 \cap \theta_2 \cap \theta_3)$	0.00	$(\theta_2 \cap \theta_3) \cup \theta_1$	0.00	$(\theta_2 \cup \theta_3) \cap \theta_1$	0.00

Table 15.7: Final results for the example of DSmC combination ($m_{1\oplus 2}$)

This is especially important for full explicit calculation of equation (15.4). As the number of possible objects and/or the number of possible sources of evidence grows, we would avoid extraordinary increase in resources needs (since the increase follows Dedekind's sequence progression in the worst case [3]).

15.3.1.1 Simple procedure for effective DSmH

Instead of viewing DSmH as a mathematical equation, we propose to view it as a procedure. Table 15.9 displays that procedure. Obviously, it is still equivalent to the mathematical equation, but this way has the advantage of being very easily understood and implemented. The ease of its implementation is due to the high resemblance of the procedure to pseudo-code, a common step in software engineering.

	$m_1(\theta_1)$ 0.8	$m_1(\theta_3)$ 0.2
$m_2(\theta_2)$ 0.9	$\theta_1 \cap \theta_2$ 0.72	$\theta_1 \cap \theta_3$ 0.18
$m_2(\theta_3)$ 0.1	$\theta_1 \cap \theta_3$ 0.08	θ_3 0.02

Table 15.8: Reduced version of table 15.6

Step S1 $(\theta_1 \cap \theta_2)$	If $(\theta_1 \cap \theta_2)$ is a constraint, then continue at step S3, otherwise, the mass $m_1(X_1)\, m_2(X_2)$ is added to the mass $A = (\theta_1 \cap \theta_2)$.
Step S3 $(\theta_1 \cup \theta_2)$	If $(\theta_1 \cup \theta_2)$ is a constraint, then continue at step S2, otherwise, the mass $m_1(X_1)\, m_2(X_2)$ is added to the mass $A = (\theta_1 \cup \theta_2)$.
Step S2 $(u(X_1) \cup u(X_2))$	If $(u(X_1) \cup u(X_2))$ is a constraint, then add mass to I_t, otherwise, the mass $m_1(X_1)\, m_2(X_2)$ is added to the mass $A = (u(X_1) \cup u(X_2))$.

Table 15.9: Procedure to apply to each pair of sets (X_1, X_2) until its combined mass is given to a set

15.3.2 Notation system used

15.3.2.1 Sum of products

The system we conceived treats information in terms of *union of intersections* or *sum of products*. The sum (ADD) is being represented by union ($\cup$), and the product (MULT) by intersection ($\cap$). We have chosen this, instead of *product of sums*, to avoid having to treat parenthesis. We could also use the principles developed for logic circuits such as Karnaugh table, Boolean rules, etc. Here are few examples of this notation:

- $\theta_1 \cap \theta_2 \cap \theta_3 = \theta_1\theta_2\theta_3 = [1, MULT, 2, MULT, 3]$
- $\theta_1 \cup \theta_2 \cup \theta_3 = \theta_1 + \theta_2 + \theta_3 = [1, ADD, 2, ADD, 3]$
- $(\theta_1 \cap \theta_2) \cup \theta_3 = \theta_1\theta_2 + \theta_3 = [1, MULT, 2, ADD, 3] = [3, ADD, 1, MULT, 2]$
- $(\theta_1 \cup \theta_2) \cap \theta_3 = (\theta_1 \cap \theta_3) \cup (\theta_2 \cap \theta_3) = \theta_1\theta_3 + \theta_2\theta_3 = [1, MULT, 3, ADD, 2, MULT, 3]$
- $(\theta_1 \cap \theta_2 \cap \theta_3) \cup (\theta_4 \cap \theta_5) = \theta_1\theta_2\theta_3 + \theta_4\theta_5 = [1, MULT, 2, MULT, 3, ADD, 4, MULT, 5]$

15.3.2.2 Conversion between *sum of products* and *product of sums* notation

As we have seen above, we will use the *sum of products* as our main way of writing sets. However, as we will later see, we will need to use the *product of sums* or intersections of unions in some parts of our system to simplify the calculation process. More specifically, this dual system of notation, introduced in the last two columns of table 15.10, was done so we would be able to use the same algorithm to work with the matrix of unions and the matrix of intersections. Table 15.10 thus presents the notation used, accompanied with its equivalent mathematical notation. We can see in the *sum of products* notation in table 15.10, that a line represents a monomial of product type (e.g. $\theta_1\theta_3$) and that lines are then summed to get unions (e.g. $\theta_1\theta_3 + \theta_2$). In the *product of sums* notation, we have the reversed situation where lines represents a monomial of sum type (e.g. $\theta_1+\theta_3$) and that lines are then multiplied to get intersections (e.g. $\theta_2\,(\theta_1+\theta_3)$).

Mathematical	Matlab input/output	Sum of products	Product of sums
$\{\theta_1\}$	$[1]$	$\begin{bmatrix}1\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
$\{\theta_1\cup\theta_2\}$	$[1, ADD, 2]$	$\begin{bmatrix}1\\2\end{bmatrix}$	$\begin{bmatrix}1 & 2\end{bmatrix}$
$\{\theta_1\cap\theta_2\}$	$[1, MULT, 2]$	$\begin{bmatrix}1 & 2\end{bmatrix}$	$\begin{bmatrix}1\\2\end{bmatrix}$
$\{(\theta_2)\cup(\theta_1\cap\theta_3)\}$	$[2, ADD, 1, MULT, 3]$	$\begin{bmatrix}2 & \\ 1 & 3\end{bmatrix}$	–
$\{(\theta_1\cup\theta_2)\cap(\theta_2\cup\theta_3)\}$	–	–	$\begin{bmatrix}1 & 2\\ 2 & 3\end{bmatrix}$
$\{(\theta_1\cap\theta_2)\cup(\theta_2\cap\theta_3)\}$	$[1, MULT, 2, ADD, 2, MULT, 3]$	$\begin{bmatrix}1 & 2\\ 2 & 3\end{bmatrix}$	–
$\{(\theta_2)\cap(\theta_1\cup\theta_3)\}$	–	–	$\begin{bmatrix}2 & \\ 1 & 3\end{bmatrix}$

Table 15.10: Equivalent notations for events

The difficult part is the conversion step from the *sum of products* to the *product of sums* notation. For the simple cases, such as the ones presented in the first three lines of table 15.10 consist only in changing matrices lines into columns and columns into lines. For simplification in the conversion process we also use the absorption rule as described in equation (15.8) which is derived from the fact that $(\theta_1\theta_2)\subseteq\theta_1$. Using that rule, we can see how came the two last rows of table 15.10 by looking at the process detailed in equations (15.9) and (15.10).

$$\theta_1+\theta_1\theta_2=\theta_1 \tag{15.8}$$

$$(\theta_1\cup\theta_2)\cap(\theta_2\cup\theta_3)=(\theta_1+\theta_2)\,(\theta_2+\theta_3)=\theta_1\theta_2+\theta_1\theta_3+\theta_2+\theta_2\theta_3=\theta_1\theta_3+\theta_2 \tag{15.9}$$

$$(\theta_1\cap\theta_2)\cup(\theta_2\cap\theta_3)=\theta_1\theta_2+\theta_2\theta_3=\theta_2\,(\theta_1+\theta_3) \tag{15.10}$$

However, in the programmed Matlab$^{\text{TM}}$ code, the following procedure is used and works for any case. It's based on the use of DeMorgan's laws as seen in equations (15.11) and (15.12). Going through DeMorgan twice let's us avoid the use of negative sets. Hence, we will still respect DSm theory even with the use of this mathematical law. The use of absorption rule, as described in equation (15.8) also helps us achieve better simplification.

$$\overline{A}\,\overline{B}=\overline{A\,+\,B} \tag{15.11}$$

$$\overline{A}\,+\,\overline{B}=\overline{A\,B} \tag{15.12}$$

Here's how we proceed for the case of conversion from a set in *sum of products* to a set in *product of sums* notation. It's quite simple actually, we begin with an inversion of operators (changing additions ($\cup$) for multiplications ($\cap$) and multiplications for additions), followed by distribution of products and a simplification step. We then end it with a second inversion of operators. Since we have used the inversion two times, we don't have to indicate the not operator, ($\overline{\overline{A}} = A$).

Let's now proceed with a short example. Suppose we want to convert the set shown in equation (15.13) to a set in product of sums notation. We proceed first as said, with an inversion of operators, which gives the set in equation (15.14). We then distribute the multiplication as we did to get the set in equation (15.15). This is then followed by a simplification giving us equation (15.16), and a final inversion of operators gives us the set in equation (15.17). The set in equation (15.17) represents the *product of sums* notation version of the set in equation (15.13), which is in *sum of products*. A simple distribution of products and simplification can get us back from (15.17) to (15.13).

$$\theta_1 + \theta_2\theta_3 + \theta_2\theta_4 \tag{15.13}$$

$$\overline{\overline{(\theta_1)}\ \left(\overline{\theta_2} + \overline{\theta_3}\right)\ \left(\overline{\theta_2} + \overline{\theta_4}\right)} \tag{15.14}$$

$$\overline{\overline{\theta_1}\,\overline{\theta_2} + \overline{\theta_1}\,\overline{\theta_2}\,\overline{\theta_4} + \overline{\theta_1}\,\overline{\theta_2}\,\overline{\theta_3} + \overline{\theta_1}\,\overline{\theta_3}\,\overline{\theta_4}} \tag{15.15}$$

$$\overline{\overline{\theta_1}\,\overline{\theta_2} + \overline{\theta_1}\,\overline{\theta_3}\,\overline{\theta_4}} \tag{15.16}$$

$$(\theta_1 + \theta_2)\,(\theta_1 + \theta_3 + \theta_4) \tag{15.17}$$

15.3.3 How simple can it be

We have completed conception of a Matlab$^{\text{TM}}$ code for the dynamic case. We've tried to optimize the code but some work is still necessary. It's now operational for a restricted body of evidence and well behaved. Here's an example of the input required by the system with the events from table 15.11. We will also proceed with θ_2 as a constraint making the following constraints too:

- $\theta_1 \cap \theta_2 \cap \theta_3$
- $\theta_1 \cap \theta_2$
- $\theta_2 \cap \theta_3$
- $(\theta_1 \cup \theta_3) \cap \theta_2$

Note that having θ_2 as a constraint, has an impact on more cases than the enumerated ones above. In fact, if we obtain cases like $\theta_1 \cup \theta_2$ for instance, since θ_2 is a constraint, the resulting case would then be θ_1. We will have to consider this when evaluating final bba for the result.

As we can see, only focal elements are transmitted to and received from the system. Moreover, these focal elements are all in *sum of products*. The output also include Belief and Plausibility values of the result.

Notice also that we have dynamic constraints capability, meaning that we can put constraints on each step of the combination. They can also differ at each step of the calculation. Instead of considering constraints only at the final step of combination, this system is thus able to

(D^{Θ})	$m_1(A)$	$m_2(A)$	$m_3(A)$
$\{\theta_1\}$	0.7	0.0	0.1
$\{\theta_2\}$	0.0	0.6	0.1
$\{\theta_3\}$	0.2	0.0	0.5
$\{\theta_1 \cup \theta_2\}$	0.0	0.0	0.3
$\{\theta_1 \cup \theta_3\}$	0.0	0.2	0.0
$\{\theta_2 \cup \theta_3\}$	0.0	0.2	0.0
$\{\theta_2 \cap \theta_3\}$	0.1	0.0	0.0

Table 15.11: Information from three sources

reproduce real data fusion conditions where constraints may vary. Three different cases are presented here, keeping the same input information but varying the constraints conditions.

Complete commented listing of the produced Matlab$^{\text{TM}}$ code is available in the appendix. For the present section, only the parameters required in input and the output are displayed.

```
% Example with dynamic constraints kept stable
% INPUT FOR THE MATLAB PROGRAM

number_sources    = 3;   kind    = ['dynamic'];
info(1).elements = {[1],[3], [2, MULT, 3]};  info(1).masses   = [0.7, 0.2, 0.1];
info(2).elements = {[2],[1, ADD, 3], [2, ADD, 3]};  info(2).masses   = [0.6, 0.2, 0.2];
info(3).elements = {[1], [2], [3], [1, ADD, 2]};  info(3).masses   = [0.1, 0.1, 0.5, 0.3];
constraint{1}    = {[2], [1, MULT, 2], [2, MULT, 3],...
                    [1, MULT, 2, ADD, 3, MULT, 2], [1, MULT, 2, MULT, 3]};
constraint{2}    = {[2], [1, MULT, 2], [2, MULT, 3],...
                    [1, MULT, 2, ADD, 2, MULT, 3], [1, MULT, 2, MULT, 3]};

% OUTPUT OF THE MATLAB PROGRAM

DSm hybrid                    Plausibility              Belief
1 : m=0.28800000               1 : m=1.00000000          1 : m=0.82000000
1 MULT 3 : m=0.53200000        1 MULT 3 : m=1.00000000   1 MULT 3 : m=0.53200000
3 : m=0.17800000               3 : m=1.00000000          3 : m=0.71000000
1 ADD 3 : m=0.00200000         1 ADD 3 : m=1.00000000    1 ADD 3 : m=1.00000000

% Example with dynamic constraints applied only once at the end
% CONSTRAINTS INPUT FOR THE MATLAB PROGRAM

constraint{1}    = {};
constraint{2}    = {[2], [1, MULT, 2], [2, MULT, 3],...
                    [1, MULT, 2, ADD, 2, MULT, 3], [1, MULT, 2, MULT, 3]};

% OUTPUT OF THE MATLAB PROGRAM

DSm hybrid                    Plausibility              Belief
1 : m=0.36800000               1 : m=1.00000000          1 : m=0.61000000
1 MULT 3 : m=0.24200000        1 MULT 3 : m=1.00000000   1 MULT 3 : m=0.24200000
3 : m=0.39000000               3 : m=1.00000000          3 : m=0.63200000

% Example with dynamic constraints varying between steps of calculation
% CONSTRAINTS INPUT FOR THE MATLAB PROGRAM
```

```
constraint{1}    = {[2, MULT, 3], [2, ADD, 3]};
constraint{2}    = {[2], [1, MULT, 2], [2, MULT, 3],...
                  [1, MULT, 2, ADD, 2, MULT, 3], [1, MULT, 2, MULT, 3]};

% OUTPUT OF THE MATLAB PROGRAM

DSm hybrid                    Plausibility               Belief
1 : m=0.31200000               1 : m=1.00000000           1 : m=0.55400000
1 ADD 3 : m=0.14800000         1 ADD 3 : m=1.00000000     1 ADD 3 : m=1.00000000
1 MULT 3 : m=0.24200000        1 MULT 3 : m=1.00000000    1 MULT 3 : m=0.24200000
3 : m=0.29800000               3 : m=1.00000000           3 : m=0.54000000
```

15.3.4 Optimization in the calculation algorithm

15.3.4.1 How does it work

Being treated by a vectorial interpreter, our Matlab$^{\text{TM}}$ code had to be adapted in consequence. We have also been avoiding, as much as we could, the use of `for` and `while` loops.

Our Matlab$^{\text{TM}}$ code was conceived with two main matrices, one containing intersections, the other one containing unions. The input information is placed into a matrix identified as the fusion matrix. When building this matrix, our program puts in a vector each unique objects that will be used, hence defining total ignorance (I_t) for the case in input. Each elements of this matrix is a structure having two fields: sets and masses. Note also that only the first row and column of the matrix is filled with the input information. The rest of the matrix will contain the result.

It is easier to proceed with the intersection between two sets A and B using *product of sums* and to proceed with the union $A \cup B$ using *sum of products*. Because of that, we have chosen to keep the intersection matrix in the *product of sums* notation and the union matrix in the *sum of products* while working on these matrices separately.

To build the union matrix, we use information from the fusion matrix with the *sum of products* notation. The intersection matrix uses the *product of sums* notation for its construction with the information from the fusion matrix. However, once the intersection matrix is built, a simple conversion to the *sum of products* notation is done as we have described earlier. This way, data from this table can be compatible with those from the fusion and the union matrices.

Once the basis of the union matrix is defined, a calculation of the content is done by evaluating the result of the union of focal elements combination $m_1(X_i)\, m_2(X_j)$. The equivalent is done with the intersection matrix, replacing the union with an intersection obviously. Once the calculation of the content of the intersection matrix completed, it is converted to the sum of product notation.

The next step consist to fill up the fusion matrix with the appropriate information depending on the presence of constraints and following the procedure described earlier for the calculation of the DSmH combination rule.

In the case we want to fuse information from more than two sources, we could choose to fuse the information dynamically or statically. The first case is being done by fusing two sources at a time. The latter case considers information from all sources at once. Note however that our code is only able to proceed with the calculation dynamically for the time being. We will now proceed step by step with a full example, interlaced with explanations on the procedure, using the information from table 15.11 and the constraints described in section 15.3.3.

Table 15.12 gives us the union result from each combination of focal elements from the first two sources of evidence. The notation used in the case for union matrices is the *sum of products*. In the case of table 15.13, the intersection matrix, it is first built in the *product of sums* notation so the same calculation algorithm can be used to evaluate the intersection result from each combination of focal elements from the first two sources of evidence as it was used in the union matrix. As we'll see, a conversion to the *sum of products* notation is done to be able to obtain table 15.14.

m_1 \ m_2	$\begin{bmatrix} 2 \end{bmatrix}$ 0.6	$\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 0.2	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.2
$\begin{bmatrix} 1 \end{bmatrix}$ 0.7	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ 0.42	$\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 0.14	$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ 0.14
$\begin{bmatrix} 3 \end{bmatrix}$ 0.2	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.12	$\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 0.04	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.04
$\begin{bmatrix} 2 & 3 \end{bmatrix}$ 0.1	$\begin{bmatrix} 2 \end{bmatrix}$ 0.06	$\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 0.02	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.02

Table 15.12: Union matrix with bba's m_1, m_2 information from table 15.11 in *sum of products* notation

m_1 \ m_2	$\begin{bmatrix} 2 \end{bmatrix}$ 0.6	$\begin{bmatrix} 1 & 3 \end{bmatrix}$ 0.2	$\begin{bmatrix} 2 & 3 \end{bmatrix}$ 0.2
$\begin{bmatrix} 1 \end{bmatrix}$ 0.7	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ 0.42	$\begin{bmatrix} 1 \end{bmatrix}$ 0.14	$\begin{bmatrix} 1 & \\ 2 & 3 \end{bmatrix}$ 0.14
$\begin{bmatrix} 3 \end{bmatrix}$ 0.2	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.12	$\begin{bmatrix} 3 \end{bmatrix}$ 0.04	$\begin{bmatrix} 3 \end{bmatrix}$ 0.04
$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.1	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.06	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.02	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.02

Table 15.13: Intersection matrix with bba's m_1, m_2 information from table 15.11 in *product of sums* notation

We obtained $\theta_2\theta_3$ as a result in the second result cell in the last row of table 15.13 because the intersection $(\theta_2 \cdot \theta_3) \cdot (\theta_1 + \theta_3)$ gives us $\theta_1\theta_2\theta_3 + \theta_2\theta_3$ which, following absorption rule, gives us $\theta_2\theta_3$. The same process occurs on the second result cell in the first row of the same table where $\theta_1 \cdot (\theta_1 + \theta_3) = \theta_1 + \theta_1\theta_3 = \theta_1$.

m_1 \ m_2	$\begin{bmatrix} 2 \end{bmatrix}$ 0.6	$\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 0.2	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.2
$\begin{bmatrix} 1 \end{bmatrix}$ 0.7	$\begin{bmatrix} 1 & 2 \end{bmatrix}$ 0.42	$\begin{bmatrix} 1 \end{bmatrix}$ 0.14	$\begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}$ 0.14
$\begin{bmatrix} 3 \end{bmatrix}$ 0.2	$\begin{bmatrix} 2 & 3 \end{bmatrix}$ 0.12	$\begin{bmatrix} 3 \end{bmatrix}$ 0.04	$\begin{bmatrix} 3 \end{bmatrix}$ 0.04
$\begin{bmatrix} 2 & 3 \end{bmatrix}$ 0.1	$\begin{bmatrix} 2 & 3 \end{bmatrix}$ 0.06	$\begin{bmatrix} 2 & 3 \end{bmatrix}$ 0.02	$\begin{bmatrix} 2 & 3 \end{bmatrix}$ 0.02

Table 15.14: Intersection matrix with bba's m_1, m_2 information from table 15.11 in *sum of products* notation

From the tables 15.12 and 15.14 we proceed with the DSmH and choose, according to constraints, from which table the result will come. We might also have to evaluate $(u(X_1) \cup u(X_2))$, or give the mass to the total ignorance if the intersection and union matrices' sets are constrained. We've displayed the choice made in the fusion matrix in table 15.15 with these symbols $\cap$ (intersection), $\cup$ (union), **u** (union of the sum of objects of combined sets), **It** (total ignorance). As you will see, we have chosen a case where we have constraints applied at each step of combination, e.g. when $[m_1, m_2]$ and when $[m_1 \oplus m_2, m_3]$ are combined.

Table 15.16 is the simplified version of table 15.15 in which sets has been adapted to consider constraints. It's followed by table 15.17 which represents the results from the first combination.

As we can see in table 15.15, the first result cell from the first row was obtained from the union matrix because $\theta_1 \cap \theta_2$ is a constraint. Also, the first result cell from the last row was obtained from the union of the sum of objects of the combined sets because $\theta_2 \cap \theta_3$ is a constraint in the intersection table (table 15.14) at the same position, so is θ_2 in the union table (table 15.12).

m_2 m_1	$\begin{bmatrix} 2 \\ \end{bmatrix}$ 0.6	$\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 0.2	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.2
$\begin{bmatrix} 1 \\ \end{bmatrix}$ 0.7	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ 0.42 $\cup$	$\begin{bmatrix} 1 \\ \end{bmatrix}$ 0.14 $\cap$	$\begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}$ 0.14 $\cap$
$\begin{bmatrix} 3 \\ \end{bmatrix}$ 0.2	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.12 $\cup$	$\begin{bmatrix} 3 \\ \end{bmatrix}$ 0.04 $\cap$	$\begin{bmatrix} 3 \\ \end{bmatrix}$ 0.04 $\cap$
$\begin{bmatrix} 2 & 3 \\ \end{bmatrix}$ 0.1	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.06 $\mathbf{u}$	$\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 0.02 $\cup$	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.02 $\cup$

Table 15.15: Fusion matrix with bba's m_1, m_2 information from table 15.11 in *sum of products* notation

m_2 m_1	$\begin{bmatrix} 2 \\ \end{bmatrix}$ 0.6	$\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 0.2	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ 0.2
$\begin{bmatrix} 1 \\ \end{bmatrix}$ 0.7	$\begin{bmatrix} 1 \\ \end{bmatrix}$ 0.42 $\cup$	$\begin{bmatrix} 1 \\ \end{bmatrix}$ 0.14 $\cap$	$\begin{bmatrix} 1 & 3 \\ \end{bmatrix}$ 0.14 $\cap$
$\begin{bmatrix} 3 \\ \end{bmatrix}$ 0.2	$\begin{bmatrix} 3 \\ \end{bmatrix}$ 0.12 $\cup$	$\begin{bmatrix} 3 \\ \end{bmatrix}$ 0.04 $\cap$	$\begin{bmatrix} 3 \\ \end{bmatrix}$ 0.04 $\cap$
$\begin{bmatrix} 2 & 3 \\ \end{bmatrix}$ 0.1	$\begin{bmatrix} 3 \\ \end{bmatrix}$ 0.06 $\mathbf{u}$	$\begin{bmatrix} 1 \\ 3 \end{bmatrix}$ 0.02 $\cup$	$\begin{bmatrix} 3 \\ \end{bmatrix}$ 0.02 $\cup$

Table 15.16: Simplified fusion matrix with bba's m_1, m_2 information from table 15.11 in *sum of products* notation

On the first row of table 15.16, the first result giving us θ_1 is obtained because $\theta_1 \cup \theta_2 = \theta_1$ when θ_2 is a constraint. The same process gave us $\theta_1 \cap \theta_3$ in the last cell of the first row. In that case, we obtained that result having $\theta_1 \cap \theta_2$ as a constraint where $\theta_1\theta_2 + \theta_1\theta_3 = \theta_1\theta_3$. Since we have more than two sources and have chosen a dynamic methodology: once the first two sources combined, we will have to proceed with a second combination. This time, we combine the results from the first combination $m_1 \oplus m_2$ with the third event from source of evidence m_3.

$[1]$	$[3]$	$[1 \cup 3]$	$[1 \cap 3]$
0.56	0.28	0.02	0.14

Table 15.17: Result of the combination of m_1 and m_2 from table 15.11

Table 15.18 represents the union matrix from second combination.

m_3 / $m_1 \oplus m_2$	$\begin{bmatrix}1\end{bmatrix}$ 0.1	$\begin{bmatrix}2\end{bmatrix}$ 0.1	$\begin{bmatrix}3\end{bmatrix}$ 0.5	$\begin{bmatrix}1\\2\end{bmatrix}$ 0.3
$\begin{bmatrix}1\end{bmatrix}$ 0.56	$\begin{bmatrix}1\end{bmatrix}$ 0.056	$\begin{bmatrix}1\\2\end{bmatrix}$ 0.056	$\begin{bmatrix}1\\3\end{bmatrix}$ 0.28	$\begin{bmatrix}1\\2\end{bmatrix}$ 0.168
$\begin{bmatrix}3\end{bmatrix}$ 0.28	$\begin{bmatrix}1\\3\end{bmatrix}$ 0.028	$\begin{bmatrix}2\\3\end{bmatrix}$ 0.028	$\begin{bmatrix}3\end{bmatrix}$ 0.14	$\begin{bmatrix}1\\2\\3\end{bmatrix}$ 0.084
$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.14	$\begin{bmatrix}1\end{bmatrix}$ 0.014	$\begin{bmatrix}1 & 3\\2 & \end{bmatrix}$ 0.014	$\begin{bmatrix}3\end{bmatrix}$ 0.07	$\begin{bmatrix}1\\2\end{bmatrix}$ 0.042
$\begin{bmatrix}1\\3\end{bmatrix}$ 0.02	$\begin{bmatrix}1\\3\end{bmatrix}$ 0.002	$\begin{bmatrix}1\\2\\3\end{bmatrix}$ 0.002	$\begin{bmatrix}1\\3\end{bmatrix}$ 0.01	$\begin{bmatrix}1\\2\\3\end{bmatrix}$ 0.006

Table 15.18: Union matrix with bba's $m_1 \oplus m_2, m_3$ information comes from tables 15.11 and 15.15 in *sum of products* notation

Table 15.19 and 15.20 are the intersection matrix with *product of sums* and *sum of products* notation respectively.

m_3 / $m_1 \oplus m_2$	$\begin{bmatrix}1\end{bmatrix}$ 0.1	$\begin{bmatrix}2\end{bmatrix}$ 0.1	$\begin{bmatrix}3\end{bmatrix}$ 0.5	$\begin{bmatrix}1 & 2\end{bmatrix}$ 0.3
$\begin{bmatrix}1\end{bmatrix}$ 0.56	$\begin{bmatrix}1\end{bmatrix}$ 0.056	$\begin{bmatrix}1\\2\end{bmatrix}$ 0.056	$\begin{bmatrix}1\\3\end{bmatrix}$ 0.28	$\begin{bmatrix}1\end{bmatrix}$ 0.168
$\begin{bmatrix}3\end{bmatrix}$ 0.28	$\begin{bmatrix}1\\3\end{bmatrix}$ 0.028	$\begin{bmatrix}2\\3\end{bmatrix}$ 0.028	$\begin{bmatrix}3\end{bmatrix}$ 0.14	$\begin{bmatrix}1 & 2\\3 & \end{bmatrix}$ 0.084
$\begin{bmatrix}1\\3\end{bmatrix}$ 0.14	$\begin{bmatrix}1\\3\end{bmatrix}$ 0.014	$\begin{bmatrix}1\\2\\3\end{bmatrix}$ 0.014	$\begin{bmatrix}1\\3\end{bmatrix}$ 0.07	$\begin{bmatrix}1\\3\end{bmatrix}$ 0.042
$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.02	$\begin{bmatrix}1\end{bmatrix}$ 0.002	$\begin{bmatrix}2 & \\1 & 3\end{bmatrix}$ 0.002	$\begin{bmatrix}3\end{bmatrix}$ 0.01	$\begin{bmatrix}1 & 2\\1 & 3\end{bmatrix}$ 0.006

Table 15.19: Intersection matrix with bba's $m_1 \oplus m_2, m_3$ information from tables 15.11 and 15.15 in *product of sums* notation

m_3 / $m_1 \oplus m_2$	$\begin{bmatrix}1\end{bmatrix}$ 0.1	$\begin{bmatrix}2\end{bmatrix}$ 0.1	$\begin{bmatrix}3\end{bmatrix}$ 0.5	$\begin{bmatrix}1\\2\end{bmatrix}$ 0.3
$\begin{bmatrix}1\end{bmatrix}$ 0.56	$\begin{bmatrix}1\end{bmatrix}$ 0.056	$\begin{bmatrix}1 & 2\end{bmatrix}$ 0.056	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.28	$\begin{bmatrix}1\end{bmatrix}$ 0.168
$\begin{bmatrix}3\end{bmatrix}$ 0.28	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.028	$\begin{bmatrix}2 & 3\end{bmatrix}$ 0.028	$\begin{bmatrix}3\end{bmatrix}$ 0.14	$\begin{bmatrix}1 & 3\\2 & 3\end{bmatrix}$ 0.084
$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.14	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.014	$\begin{bmatrix}1 & 2 & 3\end{bmatrix}$ 0.014	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.07	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.042
$\begin{bmatrix}1\\3\end{bmatrix}$ 0.02	$\begin{bmatrix}1\end{bmatrix}$ 0.002	$\begin{bmatrix}1 & 2\\1 & 3\end{bmatrix}$ 0.002	$\begin{bmatrix}3\end{bmatrix}$ 0.01	$\begin{bmatrix}1 & \\2 & 3\end{bmatrix}$ 0.006

Table 15.20: Intersection matrix with bba's $m_1 \oplus m_2, m_3$ information from tables 15.11 and 15.15 in *sum of products* notation

Finally we get table 15.21 which consists of the final fusion matrix, table 15.22 which is a simplified version of 15.21, and table 15.23 which compiles equivalent results giving us the result of the DSmH for the information from table 15.11 with same constraints applied at each step of combination.

m_3 / $m_1 \oplus m_2$	$\begin{bmatrix}1\end{bmatrix}$ 0.1	$\begin{bmatrix}2\end{bmatrix}$ 0.1	$\begin{bmatrix}3\end{bmatrix}$ 0.5	$\begin{bmatrix}1\\2\end{bmatrix}$ 0.3
$\begin{bmatrix}1\end{bmatrix}$ 0.56	$\begin{bmatrix}1\end{bmatrix}$ 0.056 $\cap$	$\begin{bmatrix}1\\2\end{bmatrix}$ 0.056 $\cup$	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.28 $\cap$	$\begin{bmatrix}1\end{bmatrix}$ 0.168 $\cap$
$\begin{bmatrix}3\end{bmatrix}$ 0.28	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.028 $\cap$	$\begin{bmatrix}2\\3\end{bmatrix}$ 0.028 $\cup$	$\begin{bmatrix}3\end{bmatrix}$ 0.14 $\cap$	$\begin{bmatrix}1 & 3\\2 & 3\end{bmatrix}$ 0.084 $\cap$
$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.14	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.014 $\cap$	$\begin{bmatrix}1 & 3\\2 & \end{bmatrix}$ 0.014 $\cup$	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.07 $\cap$	$\begin{bmatrix}1 & 3\end{bmatrix}$ 0.042 $\cap$
$\begin{bmatrix}1\\3\end{bmatrix}$ 0.02	$\begin{bmatrix}1\end{bmatrix}$ 0.002 $\cap$	$\begin{bmatrix}1\\2\\3\end{bmatrix}$ 0.002 $\cup$	$\begin{bmatrix}3\end{bmatrix}$ 0.01 $\cap$	$\begin{bmatrix}1 & \\2 & 3\end{bmatrix}$ 0.006 $\cap$

Table 15.21: Fusion matrix with bba's $m_1 \oplus m_2, m_3$ information from table 15.11 and 15.15 in *sum of products* notation

15.3.5 Performances analysis

Since no other implementation of DSmT on D^Θ is known, we don't have the possibility of comparing it. However, we are able to track the evolution of the execution time with the growth in the number of objects or the number of sources. The same can be done with the memory requirement. Until another implementation of the DSmH is written, it is the only pertinent feasible performances analysis. The program usually gives us as output the DSmH calculation results with plausibility and belief values. However, the tests we have realized were done on the DSmH alone. The code, which can be found in the appendix, had also to be modified to output time and size of variables which can undoubtedly affect time of execution and probably size required by the program.

For the measurement of the time of execution, we have only used the `tic toc` Matlab^{TM} command between each tested cases. The `clear` command, which clears variables values, was also used to prevent Matlab^{TM} from altering execution time by using already existing variables.

$m_1 \oplus m_2$ \ m_3	$\begin{bmatrix}1 & \\ & \end{bmatrix}$ 0.1	$\begin{bmatrix}2 & \\ & \end{bmatrix}$ 0.1	$\begin{bmatrix}3 & \\ & \end{bmatrix}$ 0.5	$\begin{bmatrix}1 \\ 2\end{bmatrix}$ 0.3
$\begin{bmatrix}1 & \\ & \end{bmatrix}$ 0.56	$\begin{bmatrix}1 & \\ & \end{bmatrix}$ 0.056 $\cap$	$\begin{bmatrix}1 & \\ & \end{bmatrix}$ 0.056 $\cup$	$\begin{bmatrix}1 & 3\\ & \end{bmatrix}$ 0.28 $\cap$	$\begin{bmatrix}1 & \\ & \end{bmatrix}$ 0.168 $\cap$
$\begin{bmatrix}3 & \\ & \end{bmatrix}$ 0.28	$\begin{bmatrix}1 & 3\\ & \end{bmatrix}$ 0.028 $\cap$	$\begin{bmatrix}3 & \\ & \end{bmatrix}$ 0.028 $\cup$	$\begin{bmatrix}3 & \\ & \end{bmatrix}$ 0.14 $\cap$	$\begin{bmatrix}1 & 3\\ & \end{bmatrix}$ 0.084 $\cap$
$\begin{bmatrix}1 & 3\\ & \end{bmatrix}$ 0.14	$\begin{bmatrix}1 & 3\\ & \end{bmatrix}$ 0.014 $\cap$	$\begin{bmatrix}1 & 3\\ & \end{bmatrix}$ 0.014 $\cup$	$\begin{bmatrix}1 & 3\\ & \end{bmatrix}$ 0.07 $\cap$	$\begin{bmatrix}1 & 3\\ & \end{bmatrix}$ 0.042 $\cap$
$\begin{bmatrix}1 \\ 3\end{bmatrix}$ 0.02	$\begin{bmatrix}1 & \\ & \end{bmatrix}$ 0.002 $\cap$	$\begin{bmatrix}1 \\ 3\end{bmatrix}$ 0.002 $\cup$	$\begin{bmatrix}3 & \\ & \end{bmatrix}$ 0.01 $\cap$	$\begin{bmatrix}1 & \\ & \end{bmatrix}$ 0.006 $\cap$

Table 15.22: Simplified fusion matrix version of table 15.21 in *sum of products* notation

[1]	[3]	$[1 \cap 3]$	$[1 \cup 3]$
0.288	0.178	0.532	0.002

Table 15.23: Final result of DSmH for information from table 15.11

For the size of variable measurements, we have used the `whos` command at the end of the file `hybrid.m`. The program is divided into 22 files, however the main variables are contained in `hybrid.m`. Most of the functions of the programmed system calls very few other functions one into another. We also assume that once Matlab$^{\text{TM}}$ leaves a function, it destroys all of its variables. We considered hence the memory size values obtained within `hybrid.m` a good lower estimate of the required memory size.

Note also that the tests were done on a Toshiba Satellite Pro 6100 station which has a Pentium M 4 running at 1.69 GHz, 2x512 MB of RAM PC2700, and an 80 GB hard drive running at 7200 rpm.

15.3.5.1 Execution time vs $|\Theta|$

Figure (15.1) shows us evolution of the execution time versus the cardinality of Θ for $|\Theta|$ going from 3 to 9. Since there are large number of possible testing parameters, we have chosen to perform the tests in a specific case. It consists of measuring the evolution of the execution time versus $|\Theta|$ while keeping the number of sources to 5 with the same information provided by each source for each point. Each source gives a bba with only six focal elements ($|\mathcal{K}| = 6$).

We have chosen also to put only six constraints on each point. Moreover, the constraints are dynamical and applied at each step of combination. As we can see on figure (15.1), time evolves exponentially with $|\Theta|$.

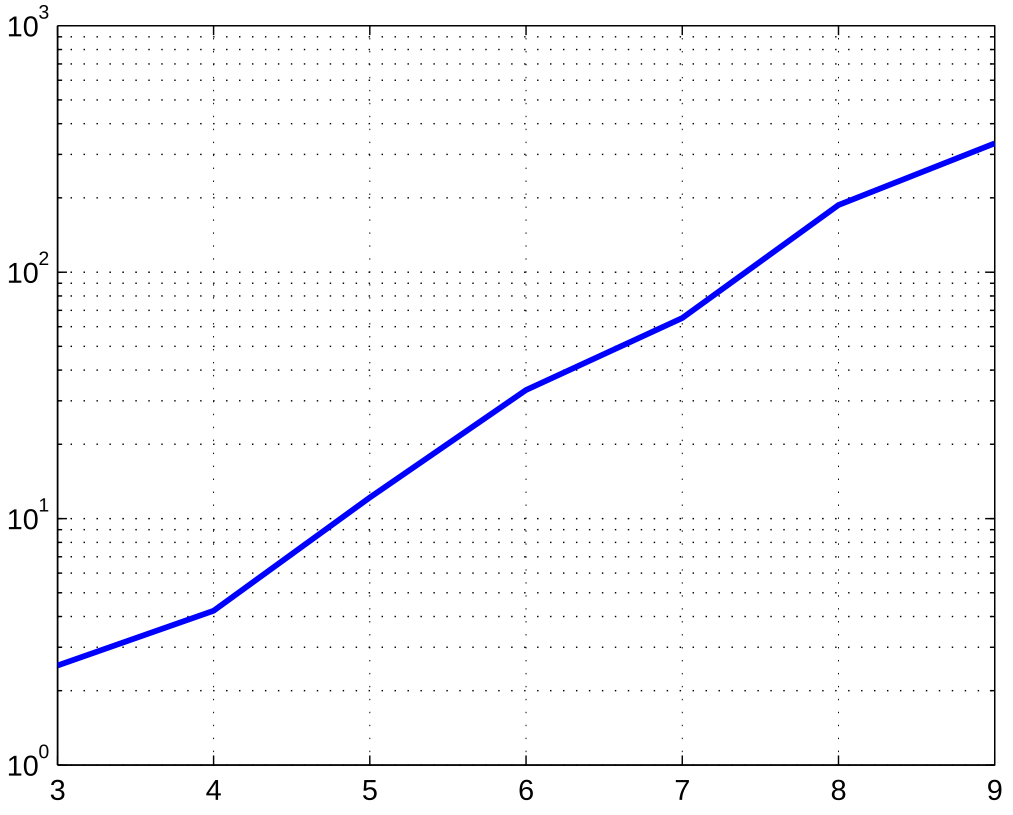

Figure 15.1: Evolution of execution time (sec) vs the cardinality of Θ

15.3.5.2 Execution time vs Number of sources

Figure (15.2) shows us the evolution of the execution time versus the number of sources going from 3 to 9. Since there are large number of possible testing parameters, we have chosen to perform the tests in a specific case. It consists of measuring the evolution of the execution time versus the number of sources while keeping $|\Theta|$ to 5 with information varying for each source for each point.

Each source gives a bba with only six focal elements ($|\mathcal{K}| = 6$). We have chosen also to put only six constraints on each point; moreover, the constraints are dynamical and applied at each step of combination. As we can see on figure (15.2), time also evolves exponentially with the number of sources.

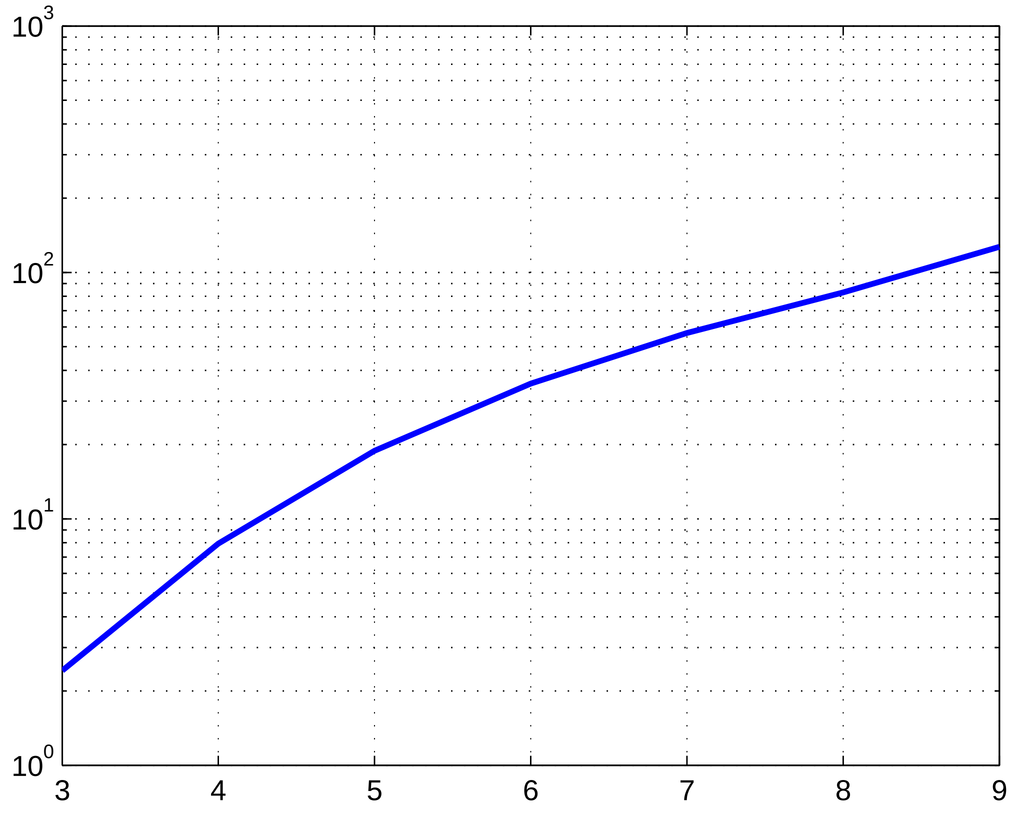

Figure 15.2: Evolution of execution time (sec) vs the number of sources

15.3.5.3 Execution time vs $|\mathcal{K}|$

Figure (15.2) shows us evolution of the execution time versus the core dimension or the number of non-zero masses going from 3 to 9. In this case, we have chosen to perform the tests while keeping $|\Theta|$ to 3 with a fixed number of sources of 5. We have chosen also to put only three constraints on each step of combination. As we can see on figure (15.3), time evolves almost linearly with the core dimension.

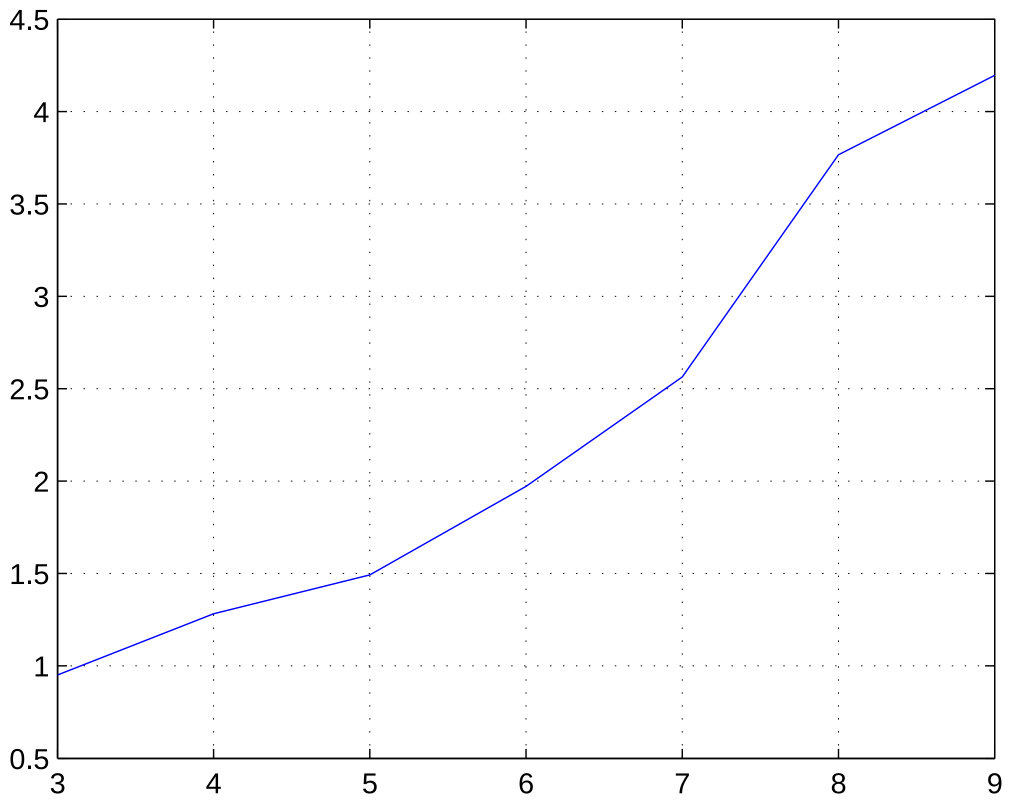

Figure 15.3: Evolution of execution time (sec) vs the cardinality of $\mathcal{K}$

15.3.5.4 Memory size vs the number of sources or $|\Theta|$

Figure (15.4) was realized under the same conditions as the input conditions for the execution time performance tests. We note that even with an increasing memory requirement, memory needs are still small. It is, of course, only the requirement for one of the many functions of our system. However, subdivisions of the code in many functions, the memory management system of Matlab$^{\text{TM}}$ and the fact that we only keep the necessary information to fuse helps keeping it at low levels. Also, during the tests we have observed in the Windows XP Pro task manager the amount of system memory used by Matlab$^{\text{TM}}$. We've noted the memory use going from 79 MB before starting the test, to a peak usage of 86 MB.

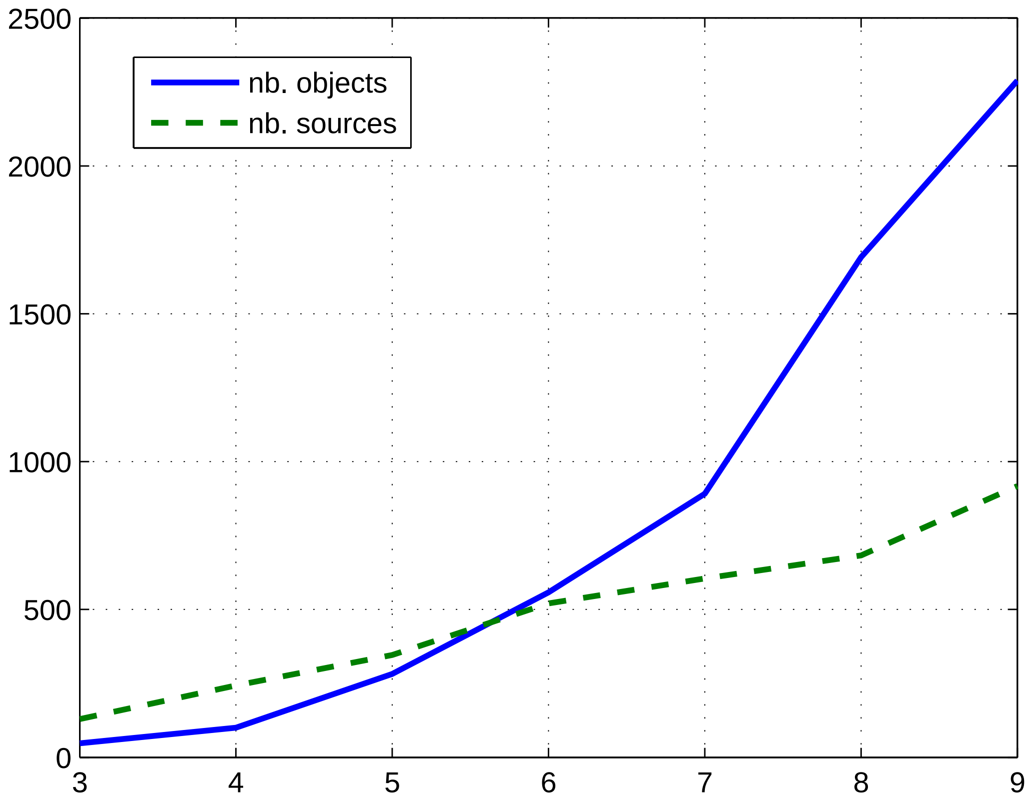

Figure 15.4: Evolution of memory size (KB) of hybrid.m workspace vs the number of sources or $|\Theta|$

We have also tried it once in static mode with a core dimension of 10 from five sources and ten constraints with three objects in the frame of refinement to see how much memory it would take. In that simple case, we went from 79 MB (before the test started) to 137 MB (a peak memory usage during the test). A huge hunger for resources was predictable for the static calculation mode with the enormous matrix it has to build with all the input information.

15.3.5.5 Further optimization to be done

Our code's algorithm is an optimization of the original DSmH calculation process. However, certain parts of our program remains to be optimized. First of all, the possibility of rejecting information and transferring it's mass to total ignorance in the case it's mass is too small or if we have too many information should be added. Second point, at many stages of our calculation, sorting is required. As we know, sorting is one of the most time consuming process in programs and it's also the case in our program. We've used two `for` loops for sorting within two other `for` loops to go through all the elements of the matrix within the file `ordre_grandeur.m`. So as the quantity of information grows, Matlab^{TM} might eventually have problems sorting the inputted information. The use of an optimized algorithm replacing this part is recommended. There's also the possibility of using the Matlab^{TM} command `sort` with some adaptations to be able to do the following sorting.

Our required sorting process in `ordre_grandeur.m` should be able to sort sets first according to the sets' size. Then, for equal sized sets, the sorting process should be able to sort in numerical order of objects. So the following set : $\theta_4 + \theta_1\theta_3\theta_4 + \theta_2\theta_3 + \theta_1\theta_3$ should be ordered this way : $\theta_4 + \theta_1\theta_3 + \theta_2\theta_3 + \theta_1\theta_3\theta_4$. A sorting process is also in use within the file `tri.m` which is able to sort matrices or sets. However the sorting process should also be optimized there.

15.4 Conclusion

As we have seen, even with the apparent complexity of DSmH, it is still possible to engineer an efficient procedure of calculation. Such a procedure enables us to conceive an efficient Matlab^{TM} code. We have conceived such a code that can perform within a reasonable amount of time by limiting the number of `for` and `while` loops exploiting Matlab's^{TM} vectorial calculation capabilities. However, even if we have obtained an optimal process of evaluating DSmH, there's still work to be done to optimize some parts of our code involving sorting.

Two avenues can be taken in the future. The first one would be to increase optimization of the actual code, trying to reduce further the number of loops, particularly in sorting. The second avenue would now be to explore how to optimize and program new combination rules such as the adaptive combination rule (ACR) [1], and the proportional conflict redistribution (PCR) rule [1].

15.5 Acknowledgements

We acknowledge Defence R&D Canada for their financial and technical supports. Pascal Djiknavorian would also like to thank his parents and his little brother, Nejmeh, Mihran, and Edouard, for encouraging him, and for their support.

15.6 References

[1] Florea M. C., Dezert J., Valin P., Smarandache F., Jousselme A.-L. , *Adaptative combination rule and proportional conflict redistribution rule for information fusion*, in Proceedings of COGIS '06 Conference, Paris, France, March 2006.

[2] Gagnon M.-L. , Grenier D., *Rapport: Fusion de données, été 2005.*, Université Laval, Canada, 2005.

[3] Smarandache F., Dezert J. (Editors), *Advances and Applications of DSmT for Information Fusion.*, Collected Works, American Research Press, Rehoboth, June 2004.

[4] Smarandache F., *Unification of Fusion Theories*, ArXiv Computer Science e-prints, arXiv:cs/0409040, 2004.

15.7 Appendix: Matlab$^{\text{TM}}$code listings

The code listed in this chapter is the property of the Government of Canada, DRDC Valcartier and has been developed by M.-L. Gagnon and P. Djiknavorian under the supervision of Dominic Grenier at Laval University, Quebec, Canada.

The code is available as is for educational purpose only. The authors can't be held responsible of any other usage. Users of the code use it at their own risks. For any other purpose, users of the code should obtain an autorization from the authors.

15.7.1 File : aff_ensemble.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description: function displaying elements and mass from a set
%
% info: elements and mass information to display
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function aff_ensemble(info)
%#inbounds
%#realonly
nI = length(info);
for k = 1 : nI
    %% displays only the non empty elements
    if ~isequal(info(k).elements,[])
    disp([num2str(info(k).elements) ' : m='  num2str(info(k).masses,'%12.8f')]);
    end
end
disp(' ')
```

15.7.2 File : aff_matrice.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description: function displaying elements and mass
%
% info: elements and mass information to display
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function aff_matrice(info)
%#inbounds
%#realonly
[m,n] = size(info);
%% go through all the objects
for k = 1 : m
    for g = 1 : n
        ensemble = info(k,g).elements
        for h = 1 : length(ensemble)
            disp([num2str(ensemble{h})]);
        end
        disp ([ 'm : ' num2str(info(k,g).masses,'%6.4f') ]);
    end
end
```

15.7.3 File : bon_ordre.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description: function ordering vectors in sets
%
% ensembleN, ensembleM : two sets in which we have to see if some values
% are identical, if so, they must be put at the same position
%
% ensembleNOut, ensembleMOut : output vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ensembleMOut, ensembleNOut] = bon_ordre(ensembleM, ensembleN)
%#inbounds
%#realonly
ensembleMOut  = {};
ensembleNOut  = {};
ensemble1     = [];
ensemble2     = [];
ensemble_temp = [];
plus_grand    = 1;
%% go through all the objects
if length(ensembleN) >= length(ensembleM)
    ensemble1  = ensembleN;
    ensemble2  = ensembleM;
    plus_grand = 1;
else
    ensemble1  = ensembleM;
    ensemble2  = ensembleN;
    plus_grand = 2;
end
%% check if there is two identical sets, otherwise check vectors
for g = 1 : length(ensemble2)
    for h = 1 : length(ensemble1)
        if isequal(ensemble1{h},ensemble2{g})
            ensemble_temp = ensemble1{g};
            ensemble1{g}  = ensemble1{h};
            ensemble1{h}  = ensemble_temp;
        end
    end
end
if isequal(plus_grand,1)
    ensembleMOut = ensemble2;
    ensembleNOut = ensemble1;
elseif isequal(plus_grand,2)
    ensembleNOut = ensemble2;
    ensembleMOut = ensemble1;
end
```

15.7.4 File : calcul_DSm_hybrid_auto.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  function to execute a DSm hybrid rule of combination
%               in dynamic or static mode
%
% Output: displayed in sum of product
%                                   sum for union
%                                   product for intersection
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte)
%#inbounds
%#realonly
global ADD
global MULT
ADD  = -2;
MULT = -1;
Iall       = [];
Ihyb       = [];
contraire  = [];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% compute the product of sum
[contraire_complet, contraire] = faire_contraire(info);
%% case with two sources
if isequal(nombre_source,2)
    Ihyb   = hybride(info, contrainte{1},contraire,2,nombre_source,contraire_complet);
    shafer = 0;
    Iall = depart(Ihyb,2);
    Ihyb = depart(Ihyb,1);
    disp('DSm hybride');
    aff_ensemble(Ihyb);
else
    %% case with more than two sources : check the type 'sorte' of DSmH
    %% case dynamic
    if isequal(sorte,'dynamique')
       Ihyb = hybride(info,contrainte{1},contraire,2,nombre_source,contraire_complet);
        for g = 3 : nombre_source
            Ihyb          = depart(Ihyb,2);
            ensemble_step = {};
            masses_step   = [];
            disp('DSm hybride');
            aff_ensemble(Ihyb)
            for h = 1 : length(Ihyb)
                ensemble_step{h} = Ihyb(h).elements;
                masses_step(h)   = Ihyb(h).masses;
            end
            info(1).elements    = {};               info(1).masses     = [];
            info(2).elements    = {};               info(2).masses     = [];
            info(1).elements    = ensemble_step;    info(1).masses     = masses_step;
            info(2)             = info(g);
            [contraire_complet, contraire] = faire_contraire(info);
            clear Ihyb;
     Ihyb = hybride(info,contrainte{g-1},contraire,2,nombre_source,contraire_complet);
```

```
        end
        %% replace numerical value of ADD and MULT by the text 'ADD','MULT'
        Iall = depart(Ihyb,2);
        Ihyb = depart(Ihyb,1);
        disp('DSm hybride');
        aff_ensemble(Ihyb);
    %% case static
    else
        Ihyb = hybride(info,contrainte{nombre_source -1},contraire,1, ...
                                           nombre_source,contraire_complet);
        %% replace numerical value of ADD and MULT by the text 'ADD','MULT'
        Iall = depart(Ihyb,2);
        Ihyb = depart(Ihyb,1);
        disp('DSm hybride');
        aff_ensemble(Ihyb);
    end
end
%% compute belief and plausibility
Isel = Iall;
fboe = {'pl' 'bel'};
for k=1:length(fboe)
    switch fboe{k}
        case 'pl'
            Pds = plausibilite(Isel,contrainte);
            disp('Plausibilite');
            Pds = depart(Pds,1);
            aff_ensemble(Pds);
        case 'bel'
            Bds = croyance(Isel);
            disp('Croyance');
            Bds = depart(Bds,1);
            aff_ensemble(Bds);
    end
end
```

15.7.5 File : calcul_DSm_hybride.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  main file to execute a DSm hybrid rule of combination
%               in dynamic or static mode
%
% Output: displayed in sum of product
%                                   sum for union
%                                   product for intersection
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;
clc;

%#inbounds
%#realonly

global ADD
global MULT
```

```
ADD  = -2;
MULT = -1;

Iall       = [];
Ihyb       = [];
info       = [];
contrainte = [];
contraire  = [];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% WRITE EVENTS AND CONSTRAINTS IN SUM OF PRODUCT NOTATION  %

% nombre_source    = 2;
% info(1).elements = {[1], [1, ADD, 2], [1, ADD, 3], [2], [2, ADD, 3], [3]};
% info(1).masses   = [0.2, 0.17, 0.33, 0.03, 0.17, 0.1];
% info(2).elements = {[1], [2], [3]};
% info(2).masses   = [0.2, 0.4, 0.4];
%%contrainte{1}    = {};
% %contrainte{1}    = {[1, MULT, 2], [1, MULT, 3], [2, MULT, 3]};
% contrainte{1}    = {[1, MULT, 2], [1, MULT, 3], [2, MULT, 3],...
%                     [1, MULT, 2,ADD, 1, MULT, 3]...
%                     [1, MULT, 2,ADD, 2, MULT, 3]...
%                     [1, MULT, 3,ADD, 2, MULT, 3]...
%                     [1, MULT, 2,ADD, 1, MULT, 3, ADD, 2, MULT, 3]};

% nombre_source    = 3;   sorte    = ['dynamique'];
% info(1).elements = {[1],[3], [2, MULT, 3]};
% info(1).masses   = [0.7, 0.2, 0.1];
% info(2).elements = {[2],[1, ADD, 3], [2, ADD, 3]};
% info(2).masses   = [0.6, 0.2, 0.2];
% info(3).elements = {[1], [2], [3], [1, ADD, 2]};
% info(3).masses   = [0.1, 0.1, 0.5, 0.3];
% contrainte{1}    = {[2, MULT, 3], [2, ADD, 3]};
% contrainte{2}    = {[2], [1, MULT, 2], [2, MULT, 3],...
%                    [1, MULT, 2, ADD, 2, MULT, 3], [1, MULT, 2, MULT, 3]};

nombre_source    = 2;
info(1).elements = {[1, MULT, 2, ADD, 1, MULT, 3, ADD, 2, MULT, 3], [1]};
info(1).masses  =   [0.6, 0.4];
info(2).elements = {[1, MULT, 2, ADD, 1, MULT, 3, ADD, 2, MULT, 3], [1]};
info(2).masses  =   [0.4, 0.6];
contrainte{1}    = {};

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% compute the product of sum
[contraire_complet, contraire] = faire_contraire(info);
%% case with two sources
if isequal(nombre_source,2)
    Ihyb   = hybride(info, contrainte{1},contraire,2,nombre_source,contraire_complet);
    shafer = 0;
    Iall = depart(Ihyb,2);
    Ihyb = depart(Ihyb,1);
    disp('DSm hybride');
    aff_ensemble(Ihyb);
else
    %% case with more than two sources : check the type 'sorte' of DSmH
    %% case dynamic
    if isequal(sorte,'dynamique')
      Ihyb = hybride(info,contrainte{1},contraire,2,nombre_source,contraire_complet);
        for g = 3 : nombre_source
            Ihyb          = depart(Ihyb,2);
            ensemble_step = {};
            masses_step   = [];
            disp('DSm hybride');
            aff_ensemble(Ihyb)
            for h = 1 : length(Ihyb)
                ensemble_step{h} = Ihyb(h).elements;
                masses_step(h)   = Ihyb(h).masses;
            end
            info(1).elements    = {};               info(1).masses  = [];
            info(2).elements    = {};               info(2).masses  = [];
            info(1).elements    = ensemble_step;    info(1).masses  = masses_step;
            info(2)             = info(g);
            [contraire_complet, contraire] = faire_contraire(info);
            clear Ihyb;
            Ihyb  = hybride(info,contrainte{g-1},contraire,2,nombre_source, ...
                                                      contraire_complet);
        end
        %% replace numerical value of ADD and MULT by the text 'ADD','MULT'
        Iall = depart(Ihyb,2);
        Ihyb = depart(Ihyb,1);
        disp('DSm hybride');
        aff_ensemble(Ihyb);
    %% case static
    else
        Ihyb = hybride(info,contrainte{nombre_source -1},contraire,1,...
                                         nombre_source,contraire_complet);
        %% replace numerical value of ADD and MULT by the text 'ADD','MULT'
        Iall = depart(Ihyb,2);
        Ihyb = depart(Ihyb,1);
        disp('DSm hybride');
        aff_ensemble(Ihyb);
    end
end
%% compute belief and plausibility
Isel = Iall;
fboe = {'pl' 'bel'};
for k=1:length(fboe)
    switch fboe{k}
        case 'pl'
            Pds = plausibilite(Isel,contrainte);
            disp('Plausibilite');
            Pds = depart(Pds,1);
            aff_ensemble(Pds);
        case 'bel'
            Bds = croyance(Isel);
            disp('Croyance');
            Bds = depart(Bds,1);
            aff_ensemble(Bds);
    end
end
```

15.7.6 File : croyance.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description: function that computes belief
%
% I : final information for which we want to compute belief
%
% croyance_complet: output giving belief values and objects
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function croyance_complet = croyance(I)
%#inbounds
%#realonly
global ADD
global MULT
ADD  = -2;
MULT = -1;
info          = [];
matrice_monome = [];
ignorance     = [];
nombreElement  = 0;
ensemble      = {};
vecteur1      = [];
vecteur2      = [];
f             = 1;
j             = 1;
%% separates objects, remove words ADD and MULT
for g = 1 : length(I)
    if ~isempty(I(g).elements)
        ensemble{f} = I(g).elements;
        vecteur1(f) = I(g).masses;
        vecteur2(f) = 1;
        f           = f + 1;
    end
end
info(1).elements = ensemble;
info(1).masses   = vecteur1;
info(2).elements = ensemble;
info(2).masses   = vecteur2;
[matrice_monome,ignorance,nombreElement] = separation(info,1);
matrice_monome                           = ordre_grandeur(matrice_monome,2);
%% produces the union matrix
matrice_intersection_contraire = intersection_matrice(matrice_monome,1);
matrice_intersection_contraire = ordre_grandeur(matrice_intersection_contraire,2);
matrice_intersection_contraire = dedouble(matrice_intersection_contraire,2);
%% Those for which union equals the monome (by lines), we add their masses.
[m,n] = size(matrice_intersection_contraire);
for g = 2 : m
```

```
    for h = 2 : n
        if isequal(matrice_intersection_contraire(g,h).elements,...
                                                matrice_monome(g,1).elements)
            resultat(j).elements = matrice_monome(g,1).elements;
            resultat(j).masses   = matrice_intersection_contraire(g,h).masses;
            j                    = j + 1;
        end
    end
end
croyance_complet = dedouble(resultat,1);
```

15.7.7 File : dedouble.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description: function that removes identical values and simplifies object
%
% matrice:  matrix to simplify, can be a set
% sorte:    indicates if input is a matrix or a set
%
% retour:   output once simplified
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [retour] = dedouble(matrice,sorte)
%#inbounds
%#realonly
global REPETE
REPETE = 0;
%% case set
if isequal(sorte,1)
    ensembleOut = [];
    j           = 1;
    %% go through elements of the set
    for g = 1 : length(matrice)
        for h = g + 1 : length(matrice)
            if isequal(matrice(h).elements,matrice(g).elements)
                matrice(h).elements = REPETE;
                matrice(g).masses   = matrice(g).masses + matrice(h).masses;
            end
        end
        if ~isequal(matrice(g).elements,REPETE) & ~isequal(matrice(g).masses,0)
            ensembleOut(j).elements = matrice(g).elements;
            ensembleOut(j).masses   = matrice(g).masses;
            j                       = j + 1;
        end
    end
    retour = ensembleOut;
%% case matrix
else
    [m,n]    = size(matrice);
    vecteur1 = [];
    vecteur2 = [];
    if m > 1
        u = 2;
        y = 2;
```

```
else
    u = 1;
    y = 1;
end
%% go through elements of the matrix
for h = u : m
    for g = y : n
        ensemble        = {};
        ensemble        = matrice(h,g).elements;
        j               = 1;
        nouvel_ensemble = {};
        %% go through all vectors of the matrix
        for k = 1 : length(ensemble)
            vecteur1 = ensemble{k};
            if ~isempty(vecteur1)
                for f = k + 1 : length(ensemble)
                    vecteur2 = ensemble{f};
                    %% check if there is two identical vectors
                    if ~isempty(vecteur2)
                        if isequal(vecteur1, vecteur2)
                            vecteur1 = REPETE;
                        else
                            %% check if a vector is included in another
                            %% 2 intersection 2union3  : remove 2union3
                            compris = 0;
                            for v = 1 : length(vecteur1)
                                for c = 1 : length(vecteur2)
                                    if isequal(vecteur1(v),vecteur2(c))
                                        compris = compris + 1;
                                    end
                                end
                            end
                            if length(vecteur1) < length(vecteur2)
                                if isequal(compris, length(vecteur1))
                                    vecteur2 = REPETE;
                                end
                            else
                                if isequal(compris, length(vecteur2))
                                    vecteur1 = REPETE;
                                end
                            end
                        end
                    ensemble{f} = vecteur2;
                    end
                end
                ensemble{k} = vecteur1;
            end
            if ~isequal(ensemble{k},REPETE)
                nouvel_ensemble{j} = ensemble{k};
                j                  = j + 1;
            end
        end
        matriceOut(h,g).elements = nouvel_ensemble;
```

```
            matriceOut(h,g).masses   = matrice(h,g).masses;
        end
    end
    matriceOut = tri(matriceOut,1);
    retour = matriceOut;
end
```

15.7.8 File : depart.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  function putting ADD and MULT
%
% ensemble_complet: set for which we want to add ADD and MULT
%       each element is a cell including vectors
%       each vector is a product and a change of vector is a sum
% sorte: to know if it has to be in numerical value or not
%
% ensemble_final: output with ADD and MULT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ensemble_final] = depart(ensemble_complet,sorte)
%#inbounds
%#realonly
global A
global M
global ADDS
global MULTS
ADDS  = ' ADD ';
MULTS = ' MULT ';
A     = -2;
M     = -1;
ensemble       = [];
ensemble_final = [];
%% go through vectors of the set
for g = 1 : length(ensemble_complet)
        ensemble = ensemble_complet(g).elements;
        for k = 1 : length(ensemble)
            %% first time
            if isequal(k,1)
                if isequal(length(ensemble{k}),1)
                    if isequal(sorte,1)
                        ensemble_final(g).elements = [num2str(ensemble{1})];
                    else
                        ensemble_final(g).elements = [ensemble{1}];
                    end
                else
                    vecteur = ensemble{k};
                    for f = 1 : length(vecteur)
                        if isequal(f,1)
                            if isequal(sorte,1)
                                ensemble_final(g).elements = [num2str(vecteur(f))];
                            else
                                ensemble_final(g).elements = [vecteur(f)];
                            end
                        else
                    if isequal(sorte,1)
                        ensemble_final(g).elements = [...
                                                ensemble_final(g).elements, ...
                                                MULTS, num2str(vecteur(f))];
                    else
                        ensemble_final(g).elements = [...
                                                ensemble_final(g).elements, ...
                                                M, vecteur(f)];
                    end
                        end
                    end
                end
            %% puts ' ADD ' since change of vector
            else
                if isequal(sorte,1)
                    ensemble_final(g).elements = ...
                     [ensemble_final(g).elements, ADDS];
                else
                    ensemble_final(g).elements = ...
                     [ensemble_final(g).elements, A];
                end

                if isequal(length(ensemble{k}),1)
                    if isequal(sorte,1)
                        ensemble_final(g).elements = ...
                         [ensemble_final(g).elements, ...
                         num2str(ensemble{k})];
                    else
                        ensemble_final(g).elements = ...
                         [ensemble_final(g).elements, ...
                         ensemble{k}];
                    end
                %% puts ' MULT '
                else
                    premier = 1;
                    vecteur = ensemble{k};
                    for f = 1 : length(vecteur)
                            if premier == 1
                                if isequal(sorte,1)
                                    ensemble_final(g).elements = ...
                                     [ensemble_final(g).elements,...
                                      num2str(vecteur(f))];
                                else
                                    ensemble_final(g).elements = ...
                                     [ensemble_final(g).elements, ...
                                     vecteur(f)];
                                end
                                premier                   = 0;
                            else
                                if isequal(sorte,1)
                                    ensemble_final(g).elements = ...
                                     [ensemble_final(g).elements, ...
```

```
                                     MULTS, num2str(vecteur(f))];
                                else
                                    ensemble_final(g).elements = ...
                                     [ensemble_final(g).elements, ...
                                     M, vecteur(f)];
                                end
                            end
                    end
                end
            end
        end
    ensemble_final(g).masses = ensemble_complet(g).masses;
end
```

15.7.9 File : DSmH_auto.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% description: file from which we can call the function version of the DSmH
%
%%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
% The examples used in this file were available in :
%''Advances and Applications of DSmT for Information Fusion''
% written by par Jean Dezert and Florentin Smarandache, 2004
%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;  clc;
info       = [];
contrainte = [];
global ADD
global MULT
ADD  = -2;
MULT = -1;

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    disp(' ');
    info       = [];
    contrainte = [];
    disp('Example 1, Page 21');

 nombre_source    = 2;
 sorte            = ['dynamique'];
 info(1).elements = {[1],[2],[3],[1, ADD, 2]};
 info(1).masses   = [0.1, 0.4, 0.2, 0.3];
 info(2).elements = {[1],[2],[3],[1, ADD, 2]};
 info(2).masses   = [0.5, 0.1, 0.3, 0.1];
 contrainte{1}    = {[1, MULT, 2, MULT, 3],[1, MULT, 2],[2, MULT, 3],...
  [1, MULT, 3],[3],[1, MULT, 3, ADD, 2, MULT, 3],...
   [1, MULT, 2, ADD, 1, MULT, 3],[1, MULT, 2, ADD, 2, MULT, 3]};


    calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);
```

```
%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

   disp(' ');
   info       = [];
   contrainte = [];
   disp('Example 5, Page 86');

 nombre_source    = 2;
 info(1).elements = {[1, MULT, 3],[3],[1, MULT, 2],[2],[1],...
                     [1, ADD, 3],[1, ADD, 2]};
 info(1).masses   = [0.1, 0.3, 0.1, 0.2, 0.1, 0.1, 0.1];
 info(2).elements = {[2, MULT, 3],[3],[1, MULT, 2],[2],[1],[1, ADD, 3]};
 info(2).masses   = [0.2, 0.1, 0.2, 0.1, 0.2, 0.2];
 contrainte{1}    = {[1, MULT, 3], [1, MULT, 2, MULT, 3],[1],...
                     [1, MULT, 2],[1, MULT, 2, ADD, 1, MULT, 3]};


   calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

   disp(' ');
   info       = [];
   contrainte = [];
   disp('Example 2, Page 90');

 nombre_source    = 2;
 info(1).elements = {[1, MULT, 2, MULT, 3],[2, MULT, 3],[1, MULT, 3],...
 [1, MULT, 3, ADD, 2, MULT, 3],[3],[1, MULT, 2],[1, MULT, 2, ADD, 2, MULT, 3],...
 [1, MULT, 2, ADD, 1, MULT, 3],[1, MULT, 2, ADD, 1, MULT, 3, ADD, 2, MULT, 3],...
 [3, ADD, 1, MULT, 2],[2],[2, ADD, 1, MULT, 3],[2, ADD, 3],[1],...
 [1, ADD, 2, MULT, 3],[1, ADD, 3],[1, ADD, 2],[1, ADD, 2, ADD, 3]};
 info(1).masses   = [0.01,0.04,0.03,0.01,0.03,0.02,0.02,0.03,0.04,...
 0.04,0.02,0.01,0.2,0.01,0.02,0.04,0.03,0.4];
 info(2).elements = {[1, MULT, 2, MULT, 3],[2, MULT, 3],[1, MULT, 3],...
 [1, MULT, 3, ADD, 2, MULT, 3],[3],[1, MULT, 2],[1, MULT, 2, ADD, 2, MULT, 3],...
 [1, MULT, 2, ADD, 1, MULT, 3],[1, MULT, 2, ADD, 1, MULT, 3, ADD, 2, MULT, 3],...
 [3, ADD, 1, MULT, 2],[2],[2, ADD, 1, MULT, 3],[2, ADD, 3],[1],...
 [1, ADD, 2, MULT, 3],[1, ADD, 3],[1, ADD, 2],[1, ADD, 2, ADD, 3]};
 info(2).masses   = [0.4,0.03,0.04,0.02,0.04,0.20,0.01,0.04,0.03,0.03,...
 0.01,0.02,0.02,0.02,0.01,0.03,0.04,0.01];
 contrainte{1}    = {[1, MULT, 2], [1, MULT, 2, MULT, 3]};


   calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

   disp(' ');
   info       = [];
   contrainte = [];
   disp('Example 7, Page 90');
 nombre_source    = 2;
 info(1).elements = {[1, MULT, 2, MULT, 3],[2, MULT, 3],[1, MULT, 3],...
 [1, MULT, 3, ADD, 2, MULT, 3],[3],[1, MULT, 2],[1, MULT, 2, ADD, 2, MULT, 3],...
 [1, MULT, 2, ADD, 1, MULT, 3],[1, MULT, 2, ADD, 1, MULT, 3, ADD, 2, MULT, 3],...
 [3, ADD, 1, MULT, 2],[2],[2, ADD, 1, MULT, 3],[2, ADD, 3],[1],...
 [1, ADD, 2, MULT, 3],[1, ADD, 3],[1, ADD, 2],[1, ADD, 2, ADD, 3]};
 info(1).masses   = [0.01,0.04,0.03,0.01,0.03,0.02,0.02,0.03,0.04,...
 0.04,0.02,0.01,0.2,0.01,0.02,0.04,0.03,0.4];
 info(2).elements = {[1, MULT, 2, MULT, 3],[2, MULT, 3],[1, MULT, 3],...
[1, MULT, 3, ADD, 2, MULT, 3],[3],[1, MULT, 2],[1, MULT, 2, ADD, 2, MULT, 3],...
[1, MULT, 2, ADD, 1, MULT, 3],[1, MULT, 2, ADD, 1, MULT, 3, ADD, 2, MULT, 3],...
[3, ADD, 1, MULT, 2],[2],[2, ADD, 1, MULT, 3],[2, ADD, 3],[1], ...
[1, ADD, 2, MULT, 3],[1, ADD, 3],[1, ADD, 2],[1, ADD, 2, ADD, 3]};
 info(2).masses   = [0.4,0.03,0.04,0.02,0.04,0.20,0.01,0.04,0.03,0.03,...
 0.01,0.02,0.02,0.02,0.01,0.03,0.04,0.01];
 contrainte{1}    = {[1, MULT, 2, MULT, 3], [2, MULT, 3], [1, MULT, 3], ...
 [1, MULT, 3, ADD, 2, MULT, 3], [3], [1, MULT, 2], ...
 [1, MULT, 2, ADD, 2, MULT, 3],  [1, MULT, 2, ADD, 1, MULT, 3], ...
 [1, MULT, 2, ADD, 1, MULT, 3, ADD, 2, MULT, 3], [3, ADD, 1, MULT, 2]};


    calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    disp(' ');
    info       = [];
    contrainte = [];
    disp('Example 3.2, Page 97');

 nombre_source    = 3;
 %sorte             = ['dynamique'];
 sorte            = ['statique'];
 info(1).elements = {[1],[2],[1, ADD, 2],[1, MULT, 2]};
 info(1).masses   = [0.1, 0.2, 0.3, 0.4];
 info(2).elements = {[1],[2],[1, ADD, 2],[1, MULT, 2]};
 info(2).masses   = [0.5, 0.3, 0.1, 0.1];
 info(3).elements = {[3],[1, MULT, 3],[2, ADD, 3]};
 info(3).masses   = [0.4, 0.3, 0.3];
 contrainte{1}    = {};
 contrainte{2}    = {[3],[1, MULT, 2, MULT, 3],[1, MULT, 3],...
 [2, MULT, 3],[1, MULT, 3, ADD, 2, MULT, 3]};


    calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    disp(' ');
    info       = [];
    contrainte = [];
    disp('Example 3.5, Pages 99-100');
 nombre_source    = 3;
 sorte            = ['dynamique'];
 %sorte            = ['statique'];
 info(1).elements = {[1],[2]};
 info(1).masses   = [0.6, 0.4];
 info(2).elements = {[1],[2]};
 info(2).masses   = [0.7, 0.3];
 info(3).elements = {[1], [2], [3]};
 info(3).masses   = [0.5, 0.2, 0.3];
 contrainte{1}    = {};
 contrainte{2}    = {[3], [1, MULT, 3], [2, MULT, 3], ...
 [1, MULT, 2, MULT, 3], [1, MULT, 3, ADD, 2, MULT, 3]};


   calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

   disp(' ');
   info       = [];
   contrainte = [];
   disp('Example 3.6, Page 100');

 nombre_source    = 2;
 info(1).elements = {[1], [2], [1, MULT, 2]};
 info(1).masses   = [0.5, 0.4 0.1];
 info(2).elements = {[1], [2], [1, MULT, 3], [4]};
 info(2).masses   = [0.3, 0.2, 0.1, 0.4];
 contrainte{1}    = {[1, MULT, 3], [1, MULT, 2], [1, MULT, 3, MULT, 4],...
  [1, MULT, 2, MULT, 3], [1, MULT, 2, MULT, 4],[1, MULT, 2, ADD, 1, MULT, 3]};


   calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

   disp(' ');
   info       = [];
   contrainte = [];
   disp('Example 5.2.1.3, Page 107');

 nombre_source    = 3;
 sorte            = ['dynamique'];
 %rep du livre = statique
 sorte            = ['statique'];
 info(1).elements = {[1],[3]};
 info(1).masses   = [0.6, 0.4];
 info(2).elements = {[2],[4]};
 info(2).masses   = [0.2, 0.8];
 info(3).elements = {[2], [4]};
 info(3).masses   = [0.3, 0.7];
 contrainte{1}    = {};
 contrainte{2}   = {[1, MULT, 3],[1, MULT, 2],[1, MULT, 4],[2, MULT, 3],...
 [2, MULT, 4],[3, MULT, 4],[1, MULT, 2, MULT, 3],[1, MULT, 2, MULT, 4],...
 [1, MULT, 3, MULT, 4],[2, MULT, 3, MULT, 4],[1, MULT, 2, MULT, 3, MULT, 4],...
 [1, MULT, 3, ADD, 1, MULT, 4],[1, MULT, 2, ADD, 1, MULT, 3],...
 [1, MULT, 2, ADD, 1, MULT, 4],[2, MULT, 3, ADD, 2, MULT, 4],...
 [1, MULT, 2, ADD, 2, MULT, 4],[1, MULT, 2, ADD, 2, MULT, 3],...
 [1, MULT, 3, ADD, 2, MULT, 3],[1, MULT, 3, ADD, 3, MULT, 4],...
 [2, MULT, 3, ADD, 3, MULT, 4],[1, MULT, 4, ADD, 2, MULT, 4],...
 [1, MULT, 4, ADD, 3, MULT, 4],[2, MULT, 4, ADD, 3, MULT, 4],...
 [1, MULT, 2, MULT, 3, ADD, 1, MULT, 2, MULT, 4],...
 [1, MULT, 3, ADD, 1, MULT, 4, ADD, 2, MULT, 3, ADD, 2, MULT, 4]};


    calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    disp(' ');
    info       = [];
    contrainte = [];
    disp('Example 5.2.2.2, Page 109');

 nombre_source    = 3;
 sorte           = ['dynamique'];
 %sorte            = ['statique'];
 info(3).elements = {[1],[2],[1, ADD, 2]};
 info(3).masses   = [0.4, 0.5, 0.1];
 info(2).elements = {[3],[4],[3, ADD, 4]};
 info(2).masses   = [0.3, 0.6, 0.1];
 info(1).elements = {[1], [1, ADD, 2]};
 info(1).masses   = [0.8, 0.2];
 contrainte{1}    = {};
 contrainte{2}    = {[1, MULT, 3],[1, MULT, 2],[1, MULT, 4],...
 [2, MULT, 3],[2, MULT, 4],[3, MULT, 4],[1, MULT, 2, MULT, 3],...
 [1, MULT, 2, MULT, 4],[1, MULT, 3, MULT, 4],[2, MULT, 3, MULT, 4],...
 [1, MULT, 2, MULT, 3, MULT, 4],[1, MULT, 3, ADD, 1, MULT, 4],...
 [1, MULT, 2, ADD, 1, MULT, 3],[1, MULT, 2, ADD, 1, MULT, 4],...
 [2, MULT, 3, ADD, 2, MULT, 4],[1, MULT, 2, ADD, 2, MULT, 4],...
 [1, MULT, 2, ADD, 2, MULT, 3],[1, MULT, 3, ADD, 2, MULT, 3],...
 [1, MULT, 3, ADD, 3, MULT, 4],[2, MULT, 3, ADD, 3, MULT, 4],...
 [1, MULT, 4, ADD, 2, MULT, 4],[1, MULT, 4, ADD, 3, MULT, 4],...
 [2, MULT, 4, ADD, 3, MULT, 4],[1, MULT, 2, MULT, 3, ADD, 1, MULT, 2, MULT, 4],...
 [1, MULT, 3, ADD, 1, MULT, 4, ADD, 2, MULT, 3, ADD, 2, MULT, 4]};


    calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    disp(' ');
    info       = [];
    contrainte = [];
    disp('Example 5.4.2, Page 116');
 nombre_source    = 3;
 %sorte            = ['dynamique'];
 sorte            = ['statique'];
 info(1).elements = {[1],[4, ADD, 5]};
 info(1).masses   = [0.99, 0.01];
 info(3).elements = {[2],[3],[4, ADD, 5]};
 info(3).masses   = [0.98, 0.01, 0.01];
 info(2).elements = {[1], [2], [3], [4, ADD, 5]};
 info(2).masses   = [0.01, 0.01, 0.97, 0.01];
 contrainte{1}    = {};
 contrainte{2}    = {};
 contrainte{2}    = {[1, MULT, 3],[1, MULT, 2],[1, MULT, 4],[2, MULT, 3],...
 [2, MULT, 4],[3, MULT, 4],[1, MULT, 2, MULT, 3],[1, MULT, 2, MULT, 4],...
 [1, MULT, 3, MULT, 4],[2, MULT, 3, MULT, 4],[1, MULT, 2, MULT, 3, MULT, 4],...
 [1, MULT, 3, ADD, 1, MULT, 4],[1, MULT, 2, ADD, 1, MULT, 3],...
 [1, MULT, 2, ADD, 1, MULT, 4],[2, MULT, 3, ADD, 2, MULT, 4],...
 [1, MULT, 2, ADD, 2, MULT, 4],[1, MULT, 2, ADD, 2, MULT, 3],...
 [1, MULT, 3, ADD, 2, MULT, 3],[1, MULT, 3, ADD, 3, MULT, 4],...
 [2, MULT, 3, ADD, 3, MULT, 4],[1, MULT, 4, ADD, 2, MULT, 4],...
 [1, MULT, 4, ADD, 3, MULT, 4],[2, MULT, 4, ADD, 3, MULT, 4],...
 [1, MULT, 2, MULT, 3, ADD, 1, MULT, 2, MULT, 4],...
 [1, MULT, 3, ADD, 1, MULT, 4, ADD, 2, MULT, 3, ADD, 2, MULT, 4],...
  [1, MULT, 5],[2, MULT, 5],[3, MULT, 5],[4, MULT, 5],[1, MULT, 2, MULT, 5],...
  [1, MULT, 3, MULT, 5],[1, MULT, 4, MULT, 5],[2, MULT, 3, MULT, 5],...
  [2, MULT, 4, MULT, 5],[3, MULT, 4, MULT, 5],[1, MULT, 2, MULT, 3, MULT, 5],...
  [1, MULT, 2, MULT, 4, MULT, 5],[1, MULT, 3, MULT, 4, MULT, 5],...
  [2, MULT, 3, MULT, 4, MULT, 5],[1, MULT, 2, MULT, 3, MULT, 4, MULT, 5],...
  [1, MULT, 4, ADD, 1, MULT, 5],[2, MULT, 4, ADD, 2, MULT, 5],...
  [3, MULT, 4, ADD, 3, MULT, 5],[1, MULT, 2, MULT, 4, ADD, 1, MULT, 2, MULT, 5],...
  [1, MULT, 3, MULT, 4, ADD, 1, MULT, 3, MULT, 5],...
  [2, MULT, 3, MULT, 4, ADD, 2, MULT, 3, MULT, 5],...
  [1, MULT, 2, MULT, 3, MULT, 4, ADD, 1, MULT, 2, MULT, 3, MULT, 5]};


    calcul_DSm_hybrid_auto(nombre_source, sorte, info, contrainte);

%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    disp(' ');
```

15.7.10 File : enlever_contrainte.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% description: function removing constraints in sets
%
% ensemble_complet:  sets composed of S1, S2, S3
% contrainte_separe: constraints' sets : divided in cells with vectors :
%                    each vector is a product, and a change of vector = sum
%
% ensemble_complet: final set
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ensemble_complet] = ...
```

```
                enlever_contrainte(ensemble_complet, contrainte_separe);

%#inbounds
%#realonly
global ENLEVE
ENLEVE = {};
ensemble_contrainte = {};
ensemble_elements   = [];
ensemble_produit    = [];

%go through contraints
for g = 1 : length(contrainte_separe)
    ensemble_contrainte = contrainte_separe{g};
    for h = 1 : length(ensemble_complet)
        %si la contrainte est en entier dans l'ensemble complet, l'enlever
        if isequal(ensemble_contrainte, ensemble_complet(h).elements)
            ensemble_complet(h).elements = ENLEVE;
            ensemble_complet(h).masses   = 0;
        end
    end
end

%go through contraints
for g = 1 : length(contrainte_separe)
    ensemble_contrainte = contrainte_separe{g};
    %si elle est un singleton
    if isequal(length(ensemble_contrainte),1) & ...
                                isequal(length(ensemble_contrainte{1}),1)

        for h = 1 : length(ensemble_complet)
            if ~isequal(ensemble_complet(h).elements, ENLEVE)
                ensemble_elements = ensemble_complet(h).elements;
                entre             = 0;
                for k = 1 : length(ensemble_elements)
                    %si une union, enleve
                    if isequal(ensemble_elements{k},ensemble_contrainte{1})
                        vecteur1             = ensemble_elements{k};
                        vecteur2             = ensemble_contrainte{1};
                        ensemble_elements{k} = setdiff(vecteur1, vecteur2);
                        entre                = 1;
                    end
                end

                if isequal(entre, 1)
                    j                     = 1;
                    ensemble_elements_new = [];
                    for k = 1 : length(ensemble_elements)
                        if ~isequal(ensemble_elements{k},[]);
                            ensemble_elements_new{j} = ensemble_elements{k};
                            j                        = j + 1;
                        end
                    end
                    ensemble_elements = [];
                    ensemble_elements = ensemble_elements_new;
                end
                ensemble_complet(h).elements = ensemble_elements;
            end
        end

    %otherwise, its an intersection
    elseif length(ensemble_contrainte) == 1

        ensemble_produit = ensemble_complet;
        for t = 1 : length(ensemble_produit)
            ensemble       = ensemble_produit(t).elements;
            j              = 1;
            entre          = 1;
            nouvel_ensemble = {};
            for h = 1 : length(ensemble)
                for y = 1 : length(ensemble_contrainte)
                    if isequal(ensemble{h}, ensemble_contrainte{y})
                        ensemble{h} = [];
                        entre       = 0;
                    else
                        nouvel_ensemble{j} = ensemble{h};
                        j                  = j + 1;
                    end
                end
            end
            ensemble_produit(t).elements = nouvel_ensemble;
            ensemble_complet(t).elements = ensemble_produit(t).elements;
        end
    end
end

%remove empty
for r = 1 : length(ensemble_complet)
    ensemble1       = ensemble_complet(r).elements;
    j               = 1;
    nouvel_ensemble = [];
    for s = 1 : length(ensemble1)
        if ~isequal(ensemble1{s},[])
            nouvel_ensemble{j} = ensemble1{s};
            j                  = j + 1;
        end
    end
    ensemble_complet(r).elements = nouvel_ensemble;
end

%combines identical elements
ensemble_complet = dedouble(ensemble_complet,2);
ensemble_complet = dedouble(ensemble_complet,1);
```

15.7.11 File : ensemble.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Description:  function regrouping equal structure from matrix
%
% matrice: the matrix to regroup
%
% ensembleOut: outputs the structure with sets of regrouped matrix
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ensembleOut] = ensemble(matrice)
%#inbounds
%#realonly
ensembleOut = [];
[m,n]       = size(matrice);
j           = 1;

if ~(m < 2)
    if isequal(matrice(2,2).elements, [])
        u = 1;
        y = 1;
    else
        u = 2;
        y = 2;
    end
else
        u = 1;
        y = 1;
end
%% go through all sets of the matrix, put the equal ones togheter and sum
%% their mass
for g = u : m
    for h = y : n
        if isequal(g,u) & isequal(h,y) & ~isequal(matrice(g,h).elements,[])
            ensembleOut(j).elements = matrice(g,h).elements;
            ensembleOut(j).masses   = matrice(g,h).masses;
            j                       = j + 1;
        elseif ~isequal(matrice(g,h).elements,[])
            compris = 0;
            for f = 1 : length(ensembleOut)
                if isequal(matrice(g,h).elements, ensembleOut(f).elements)
                    ensembleOut(f).masses = ...
                          ensembleOut(f).masses + matrice(g,h).masses;
                    compris               = 1;
                end
            end
            if isequal(compris,0)
                 ensembleOut(j).elements = matrice(g,h).elements;
                 ensembleOut(j).masses   = matrice(g,h).masses;
                 j                       = j + 1;
            end
        end
    end
end
```

15.7.12 File : faire_contraire.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  function that changes the sum of products in product of
%               sums with ADD and MULT
%
% info:         set that we want to modify
%
% ensembleOut:  once in product of sums and in same format as the input
% contraire:    only the first two information
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ensembleOut, contraire] = faire_contraire(info)
%#inbounds
%#realonly
ensembleOut = [];
j           = 1;
f           = 1;
temp        = [];
flag        = 3;
contraire   = [];
%% puts the sets in product of sums
[temp, ignorance, nombre] = separation(info,2);
temp                      = produit_somme_complet(temp);
temp                      = depart(temp,2);
%% puts back the sets in one set
for g = 1 : length(nombre)
    debut           = 1;
    d               = 1;
    ensembleElement = {};
    for h = 1 : nombre(g)
        if isequal(debut,1)
            ensembleElement{d}    = [temp(f).elements];
            ensembleOut(j).masses = temp(f).masses;
            debut                 = 0;
        else
            ensembleElement{d}    = [temp(f).elements];
            ensembleOut(j).masses = [ensembleOut(j).masses, temp(f).masses];
        end
        f   = f + 1;
        d   = d + 1;
    end
    %% ensembleOut: output, once in product of sums
    ensembleOut(j).elements = ensembleElement;
    %% contraire: only the first two elements of output
    if j < 3
        contraire(j).elements = ensembleOut(j).elements;
        contraire(j).masses   = ensembleOut(j).masses;
    end
    j = j + 1;
end
```

15.7.13 File : hybride.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:      function that executes the three steps of the DSmH
%
% info:             informations from the sources in product of sums
% contrainte:       contraints in sum of product
% contraire:        informations from sources in sum of products
%
% sorte:                indicates the type of fusion: dynamic ou static
% nombre_source:        number of source of evidence
% contraire_complet:    All the information in product of sum
%
% ensemble_complet:     final values (objects + masses)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ensemble_complet] = ...
        hybride(info,contrainte,contraire,sorte,nombre_source,contraire_complet)
matrice_intersection = [];  matrice_union         = [];  matrice_monome       = [];
ensemble_step1       = [];  ensemble_step2        = [];  ensemble_step3       = [];
ensemble_complet     = [];  vecteur_singleton     = [];  contrainte_produit   = [];
ignorance            = [];  ensemble_complet_temp = [];
%% case static
if isequal(sorte,1)
    matrice_infos           = [];
    matrice_infos_contraire = [];
    for g = 1 : nombre_source
        [matrice_infos,ignorance,nombreElement]           = ...
                              separation_unique(info(g),matrice_infos);
        [matrice_infos_contraire,ignorance,nombreElement] = ...
               separation_unique(contraire_complet(g),matrice_infos_contraire);
    end
    %% compute the intersection matrix
    matrice_intersection = intersection_matrice(matrice_infos_contraire,2);
    matrice_intersection = somme_produit_complet(matrice_intersection);
    matrice_intersection = dedouble(matrice_intersection,2);
    matrice_intersection = ordre_grandeur(matrice_intersection,1);
    %% compute the union matrix
    matrice_intersection_contraire = intersection_matrice(matrice_infos,2);
    matrice_intersection_contraire = dedouble(matrice_intersection_contraire,2);
%% case dynamic
else
    %% Separates products of each objects, also computes total ignorance
    [matrice_monome,ignorance1,nombreElement]           = separation(info,1);
    [matrice_monome_contraire,ignorance2,nombreElement] = separation(contraire,1);
    ignorance                                           = [ignorance1];
    %% compute the union matrix
    matrice_intersection_contraire = intersection_matrice(matrice_monome,1);
    matrice_intersection_contraire = ...
                              ordre_grandeur(matrice_intersection_contraire,2);
    matrice_intersection_contraire = dedouble(matrice_intersection_contraire,2);
    %% compute the intersection matrix
    matrice_intersection = intersection_matrice(matrice_monome_contraire,1);
    matrice_intersection = somme_produit_complet(matrice_intersection);
    matrice_intersection = dedouble(matrice_intersection,2);
end
%% separates objects in contraints: will help compare the with intersection
if ~isempty(contrainte)
    [contrainte_separe, ignorance3, nombre] = separation(contrainte,3);
    contrainte_separe                       = tri(contrainte_separe,2);
end
%% compute S1, S2, S3
%% If there is no constraints, simply take S1
if isempty(contrainte)
    ensemble_complet = ensemble(matrice_intersection);
%% Otherwise, we have to go throught the three steps
else
    %% Go through intersection matrix, if objects = contraints, take union,
    %% if objects from union = contraints, take union of objects, if it's a
    %% contraints, take total ignorance.
    j     = 1;  source = 1;
    [m,n] = size(matrice_intersection);
    ss = 1:m; s = 1;
    gg = 1:n; g = 1;
    %% Go through each line (s) of the matrix process by accessing each
    %% objects, by column (g)
    while s ~= (length(ss)+1)
    while g ~= (length(gg)+1)
        %% take value from intersection matrix
        ensemble_step = matrice_intersection(s,g).elements;
        %% if the flag is not active, set it to '1'
        if ~(source > 10)
            source = 1;
        end
        %% Proceed if there is something at (s,g) matrix position
        if ~isequal(ensemble_step, [])
            intersection = 0;
            for h = 1 : length(contrainte_separe)
                %% If value from intersection matrix is equal to actual
                %% constraint and if it hasn't been equal to a previous
                %% constraint, OR, if the flag was active, then proceed to
                %% union matrix.
                if (isequal(contrainte_separe{h},ensemble_step) &...
                                    isequal(intersection,0)) | isequal(source,22)
                    intersection  = 1; union = 0;
                    ensemble_step = [];
                    ensemble_step =  matrice_intersection_contraire(s,g).elements;
                    %% if the flag is not active for the union of objects
                    %% or to total ignorance, set it to '2'
                    if ~(source > 22)
                        source = 2;
                    end
                    for t = 1 : length(contrainte_separe)
                        %% If value from union matrix is equal to actual
                        %% constraint and if it hasn't been equal to a
                        %% previous constraint, OR, if the flag was active,
                        %% then proceed to union of objects calculation.
```

```
if (isequal(contrainte_separe{t},ensemble_step)&...
             isequal(union,0))  | isequal(source,33)
    union = 1; subunion = 0;
    nouveau_vecteur = [];
    ensemble_step   = {};
        ensemble1 = matrice_monome(s,1).elements;
        ensemble2 = matrice_monome(1,g).elements;
        b = 1;
        for f = 1 : length(ensemble1)
        vecteur = ensemble1{f};
            for d = 1 : length(vecteur)
                nouveau_vecteur{b} = [vecteur(d)];
                b                  = b + 1;
            end
        end
        for f = 1 : length(ensemble2)
            vecteur = ensemble2{f};
            for d = 1 : length(vecteur)
                nouveau_vecteur{b} = [vecteur(d)];
                b                  = b + 1;
            end
        end
        %% remove repetition
        for f = 1 : length(nouveau_vecteur)
            for r = f + 1 : length(nouveau_vecteur)
      if isequal(nouveau_vecteur{f},nouveau_vecteur{r})
                    nouveau_vecteur{r} = [];
                end
            end
        end
        y = 1;
        for r = 1 : length(nouveau_vecteur)
            if ~isequal(nouveau_vecteur{r},[])
                ensemble_step{y} = nouveau_vecteur{r};
                y                = y + 1;
            end
        end
         %% ordering
        matrice              = [];
        matrice(1,1).elements = ensemble_step;
        matrice(1,1).masses   = 0;
        matrice(2,2).elements = [];
        matrice     = ordre_grandeur(matrice,2);
        ensemble_step         = [];
        ensemble_step         = matrice(1,1).elements;
        %% if the flag is not active for ignorance
        if ~(source > 33)
            source = 3;
        end
    for r = 1 : length(contrainte_separe)
        %% If value from union of objects matrix is
        %% equal to actual constraint and if it
        %% hasn't been equal to previous constraint
```

```
                            %% OR, if the flag was active.
                            if (isequal(contrainte_separe{r}, ensemble_step)...
                                & isequal(subunion,0)) | isequal(source,44)
                                subunion = 1;
                                ensemble_step = {};
                                ensemble_step = ignorance;
                                source = 4;
                            end
                        end
                     end
                 end
              end
           end
           ensemble_complet_temp = [];
           ensemble_complet_temp(1).elements   = ensemble_step;
           ensemble_complet_temp(1).masses = matrice_intersection(s,g).masses;
           %% remove constraints of composed objects, if there is any
           ensemble_step_temp = ...
              enlever_contrainte(ensemble_complet_temp,contrainte_separe);
           %% once the contraints are all removed, check if the object are
           %% empty. If not, increment output matrix position, if it is
           %% empty, activate the flag following the position from where
           %% the answer would have been taken and restart loop without
           %% incrementing (s,g) intersection matrix position.
           if ~isempty(ensemble_step_temp(1).elements)
               ensemble_step = [];
               ensemble_step = ensemble_step_temp(1).elements;
               ensemble_complet(j).elements = ensemble_step;
               ensemble_complet(j).masses  = ...
                                          matrice_intersection(s,g).masses;
               ensemble_complet             = tri(ensemble_complet,1);
               j                            = j + 1;

           else
               switch (source)
                   %% CASE 4 is not used here. It's the case where there
                   %% would be a constraint on total ignorance.
                   case 1
                      source = 22;
                   case 2
                      source = 33;
                   case 3
                      source = 44;
               end
               %% Will let the while loop repeat process for actual (s,g)
               g = g - 1;
           end
       end %% 'end' for the "if ~isequal(ensemble_step, [])" line
       %% move forward in the intersection matrix
       g = g + 1;
   end %g = 1 : n (columns of intersection matrix)

   %% move forward in the intersection matrix
```

```
    s = s + 1;
    g = 1;
    end %s = 1 : m (lines of intersection matrix)
end
g = 1; s = 1;
%% Sort the content of the output matrix
ensemble_complet = tri(ensemble_complet,1);
%% Filter the ouput matrix to merge equal cells
ensemble_complet = dedouble(ensemble_complet,1);
```

15.7.14 File : intersection_matrice.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  function that computes the intersection matrix and masses
%
% sorte:            type of fusion [static | dynamic]
% matrice_monome:   initial information, once separated by objects with ADD
% and MULT removed. vector represents products, a change of vector the sum
% includes only the first line and column of the matrix
%
% matrice_intersection: return the result of intersections
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [matrice_intersection] = intersection_matrice(matrice_monome,sorte)
%% case dynamic
if isequal(sorte,1)
    matrice_intersection = [];
    [m,n]                = size(matrice_monome);
    ensembleN            = {};
    ensembleM            = {};

    %% go through the first line and column, fill the intersection matrix
    for g = 2 : m
        ensembleM = matrice_monome(g,1).elements;
        for h = 2 : n
            ensembleN                           = matrice_monome(1,h).elements;
            matrice_intersection(g,h).elements = [ensembleN,ensembleM];
            matrice_intersection(g,h).masses   = ...
                  matrice_monome(g,1).masses * matrice_monome(1,h).masses;
        end
    end
    matrice_intersection = dedouble(matrice_intersection,2);
    matrice_intersection = ordre_grandeur(matrice_intersection,2);
%% case static
else
    matrice_intersection  = [];
    matrice_intermediaire = [];
    [m,n]                 = size(matrice_monome);
    ensembleN             = {};
    ensembleM             = {};
    j                     = 1;
    s                     = 1;
    %% fill the intersection matrix by multipliying all at once
    for g = 1 : n
```

```
        ensembleM = matrice_monome(1,g).elements;
        if ~isequal(ensembleM,[])
            for h = 1 : n
                    ensembleN = matrice_monome(2,h).elements;
                    if ~isequal(ensembleN,[])
                        matrice_intermediaire(j,s).elements = [ensembleN,ensembleM];
                        matrice_intermediaire(j,s).masses   = ...
                           matrice_monome(2,h).masses * matrice_monome(1,g).masses;
                        s                                   = s + 1;
                    end
            end
        end
    end
    [r,t] = size(matrice_intermediaire);
    s     = 1;
    for g = 3 : m
        for h = 1 : t
            ensembleM = matrice_intermediaire(1,h).elements;
            for u = 1 : n
                ensembleN = matrice_monome(g,u).elements;
                if ~isequal(ensembleN,[])
                    matrice_intersection(1,s).elements = [ensembleN,ensembleM];
                    matrice_intersection(1,s).masses   = ...
                  matrice_intermediaire(1,h).masses * matrice_monome(g,u).masses;
                    s                                  = s + 1;
                end
            end
        end
        matrice_intermediaire =  matrice_intersection;
        matrice_intersection  = [];
        [r,t] = size(matrice_intermediaire);
        s = 1;
    end
    matrice_intersection = matrice_intermediaire;
    matrice_intersection = dedouble(matrice_intersection,2);
end
```

15.7.15 File : ordre_grandeur.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  function that orders vectors
%
% matrice:      matrix in which we order the vectors in the sets
%
% matriceOut:   output ordered matrix
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [matriceOut] = ordre_grandeur(matrice,sorte)
[m,n]        = size(matrice);
ensemble     = {};
ensembleTemp = [];
%% case static
if isequal(sorte,1)
    u = 1;
    y = 1;
%% case dynamic
else
    essai = matrice(2,2).elements;
    if isempty(essai)
        u = 1;
        y = 1;
    else
        u = 2;
        y = 2;
    end
end
%% Order by size vector of sets of matrix
for g = u : m
    for h = y : n
        ensemble = matrice(g,h).elements;
        for f = 1 : length(ensemble)
            for k = f + 1 : length(ensemble)
                if length(ensemble{k}) < length(ensemble{f})
                    ensembleTemp = ensemble{f};
                    ensemble{f}  = ensemble{k};
                    ensemble{k}  = ensembleTemp;
                elseif isequal(length(ensemble{k}), length(ensemble{f}))
                    vecteur1 = ensemble{k};
                    vecteur2 = ensemble{f};
                    changer  = 0;
                    for t = 1 : length(vecteur1)
                        if (vecteur1(t) < vecteur2(t)) & isequal(changer,0)
                           ensembleTemp = ensemble{f};
                            ensemble{f} = ensemble{k};
                            ensemble{k} = ensembleTemp;
                            changer     = 1;
                            break;
                        end
                    end
                end
            end
        end
        matriceOut(g,h).elements = ensemble;
        matriceOut(g,h).masses   = matrice(g,h).masses;
    end
end
```

15.7.16 File : plausibilite.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  function that calculates plausibility
%
% I:            final information for which we want plausibility
% contrainte:   initial constraints
%
% plausibilite_complet: returns plausibility and masses
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function plausibilite_complet = plausibilite(I,contrainte)
%#inbounds
%#realonly
global ADD
global MULT
ADD  = -2;
MULT = -1;
ensemble_complet   = {};
contrainte_compare = {};
info               = [];
matrice_monome     = [];
ignorance          = [];
ensemble_elements  = [];
vecteur1           = [];
vecteur2           = [];
nombreElement      = 0;
f                  = 1;
j                  = 1;
r                  = 1;
%% separates the objects, removes ADD and MULT
for g = 1 : length(I)
    if ~isempty(I(g).elements)
        ensemble_elements{f} = I(g).elements;
        vecteur2(f)          = I(g).masses;
        vecteur1(f)          = 1;
        f                    = f + 1;
    end
end
info(1).elements = ensemble_elements;
info(2).elements = ensemble_elements;
info(1).masses   = vecteur1;
info(2).masses   = vecteur2;
[matrice_monome,ignorance,nombreElement]           = separation(info,1);
[contraire_complet, contraire]                     = faire_contraire(info);
[matrice_monome_contraire,ignorance,nombreElement] = separation(contraire,1);
%% creates the intersection matrix
matrice_intersection = intersection_matrice(matrice_monome_contraire,1);
matrice_intersection = somme_produit_complet(matrice_intersection);
matrice_intersection = dedouble(matrice_intersection,2);
%% takes the contraint in sum of products, however, if there's none, do
%% nothing and put it all to '1'
entre = 0;
s     = 1;
for r = 1 : length(contrainte)
    if ~isempty(contrainte) & ~isempty(contrainte{r}) & isequal(entre,0)
        for g = 1 : length(contrainte)
            if ~isequal(contrainte{g},{})
                [contrainte_compare{s}, ignorance, nombre] = ...
                                         separation(contrainte{g},3);
                s                                          = s + 1;
            end
        end
        %% remove contraints on the intersection matrix
        [m,n] = size(matrice_intersection);
        for g = 2 : n
            ensemble_complet                            = [];
            matrice_intersection_trafique =  matrice_intersection(:,g);
            matrice_intersection_trafique(2,2).elements = [];
            ensemble_complet            = ensemble(matrice_intersection_trafique);
            ensemble_complet            = tri(ensemble_complet,1);
            ensemble_complet             = dedouble(ensemble_complet,1);
            for t = 1 : length(contrainte_compare)
                ensemble_complet = enlever_contrainte(ensemble_complet,...
                                                     contrainte_compare{t});
            end
            resultat(j).masses = 0;
            for t = 1 : length(ensemble_complet)
                if ~isempty(ensemble_complet(t).elements)
                    resultat(j).masses = resultat(j).masses + ...
                                       ensemble_complet(t).masses;
                end
            end
            resultat(j).elements = matrice_monome(g,1).elements;
            j = j + 1;
        end
        entre = 1;
    %% if there's no constraints, put it all to '1',
    elseif isequal(length(contrainte),r) & isequal(entre,0)
        [m,n] = size(matrice_monome);
        for g = 1 : m
            resultat(j).elements = matrice_monome(g,1).elements;
            resultat(j).masses   = 1;
            j                    = j + 1;
        end
    end
end
plausibilite_complet = dedouble(resultat,1);
```

15.7.17 File : produit_somme_complet.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description: function that converts input in product of sums
%
% ensemble_complet:     matrix in sum of products
%
% ensemble_produit:     matrix in product of sums
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ensemble_produit] = produit_somme_complet(ensemble_complet);
global ENLEVE
ENLEVE = {};
ensemble_elements = {}; ensemble_produit  = {};
vecteur           = []; matrice           = [];
p                 = 1;  y                 = 1;
%% go through all sets, puts them in product of sums
for g = 1 : length(ensemble_complet)
    if ~isequal(ensemble_complet(g).elements, ENLEVE)
```

```
ensemble_elements = ensemble_complet(g).elements;
if (length(ensemble_elements) >= 2)
    i                             = 1;
    ensemble_produit(p).elements = {};
    changer                       = 0;
    if length(ensemble_elements) >= 3
        vecteur1 = ensemble_elements{1};
        vecteur2 = ensemble_elements{2};
        if ~(length(vecteur1) > 1 & length(vecteur2) > 1)
            ensemble_produit(p).elements = ensemble_complet(g).elements;
            ensemble_produit(p).masses   = ensemble_complet(g).masses;
            p                            = p + 1;
        else
            changer = 1 ;
        end
    else
        changer = 1;
    end
    if isequal(changer, 1)
        for k = 1 : length(ensemble_elements) - 1
            if (k < 2)
                if (k + 1) > length(ensemble_elements)
                    x = length(ensemble_elements);
                else
                    x = k + 1;
                end
                for w = k : x
                    vecteur = ensemble_elements{w};
                    j       = 1;
                    for f = 1 : length(vecteur)
                        if isequal(length(vecteur),1)
                            ensembleN{j} = [vecteur];
                        else
                            ensembleN{j} = [vecteur(f)];
                            j            = j  + 1;
                        end
                    end
                    if isequal(i,1)
                        matrice(1,2).elements = ensembleN;
                        matrice(1,2).masses   = 0;
                        ensembleN             = {};
                        i                     = 2;
                    elseif isequal(i,2)
                        matrice(2,1).elements = ensembleN;
                        matrice(2,1).masses   = 0;
                        ensembleN             = {};
                        i                     = 1;
                     end
                end
             elseif (k >= 2) & (length(ensemble_elements) > 2)
                w = k + 1;
                j = 1;
                vecteur = ensemble_elements{w};
```

```
                    for f = 1 : length(vecteur)
                        if isequal(length(vecteur),1)
                           ensembleN{j} = [vecteur];
                        else
                           ensembleN{j} = [vecteur(f)];
                           j            = j  + 1;
                        end
                    end
                    matrice(1,2).elements = ensemble_produit(p).elements;
                    matrice(1,2).masses   = 0;
                    matrice(2,1).elements = ensembleN;
                    matrice(2,1).masses   = 0;
                    ensembleN             = {};
                end
                resultat = union_matrice(matrice);
                [s,t]    = size(resultat);
                for r = 1 : s
                    for d = 1 : t
                        masse = resultat(r,d).masses;
                        if isequal(masse, 0)
                            ensemble_produit(p).elements = ...
                                                resultat(r,d).elements;
                            ensemble_produit(p).masses   = ...
                                                ensemble_complet(g).masses;
                        end
                    end
                end
            end
            p = p + 1;
        end
    elseif isequal(length(ensemble_elements),1)
        for k = 1 : length(ensemble_elements)
            vecteur = ensemble_elements{k};
            j       = 1;
            for f = 1 : length(vecteur)
                if isequal(length(vecteur),1)
                    ensembleN{j} = [vecteur];
                else
                    ensembleN{j} = [vecteur(f)];
                    j            = j  + 1;
                end
            end
        end
        ensemble_produit(p).elements = ensembleN;
        ensembleN                    = {};
        ensemble_produit(p).masses   = ensemble_complet(g).masses;
        p                            = p + 1;
    elseif ~isequal(ensemble_elements, [])
        ensemble_produit(p).elements = ensemble_complet(g).elements;
        ensemble_produit(p).masses   = ensemble_complet(g).masses;
        p                            = p + 1;
    end
end
```

```
end
```

15.7.18 File : separation.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  separates products in input data
%
% info:         information from sources (initial data)
% sorte:        type of separation
%
% retour:       separated data (products)
% ignorance:    total ignorance
% nombreElement:number of vectors in sets of each information
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [retour,ignorance_nouveau,nombreElement] = separation(info,sorte)
global ADD
global MULT
global SOURCE
ADD    = -2;
MULT   = -1;
SOURCE = 2;
nouvelle_info     = []; %struc elements: set of vector
ensemble_monome   = []; %cell (1,1) of matrix is empty
matrice_monome    = []; %cell (1,1) of matrix is empty
retour            = [];
ignorance_nouveau = [];
%% takes each elements of each sources and separates the products
[m,n] = size(info);
if ~isequal(sorte,3)
    for g = 1 : n
        nombreElement(g) = length(info(g).elements);
    end
else
    nombreElement(1) = 1;
end
%% case dynamic or two sources
if isequal(sorte,1)
    %% variables
    ligne         = 1;
    colonne       = 2;
    ignorance     = [];
    %% go through each sources
    for g = 1 : n
        ensemble      = info(g).elements;
        vecteur_masse = info(g).masses;
        if isequal(g,SOURCE)
            colonne = 1;
            ligne   = 2;
        end
        %% go through each set of elements
        for h = 1 : length(ensemble)
            vecteur         = ensemble{h};
            nouveau_vecteur = [];
            nouvel_ensemble = {};
            k               = 1;
             %% go through each element of the vector
            %%     to separate the products and sums
            for j = 1 : length(vecteur)
                if ~isequal(vecteur(j), ADD)
                    if ~isequal(nouveau_vecteur, []) & ~isequal(vecteur(j), MULT)
                        nouveau_vecteur = [nouveau_vecteur, vecteur(j)];
                        if isequal(j,length(vecteur))
                           nouvel_ensemble{k} = nouveau_vecteur;
                           ignorance          = [ignorance, nouveau_vecteur];
                        end
                    elseif ~isequal(vecteur(j), MULT)
                        nouveau_vecteur = [vecteur(j)];
                        if isequal(j,length(vecteur))
                           nouvel_ensemble{k} = nouveau_vecteur;
                           ignorance          = [ignorance, nouveau_vecteur];
                        end
                    end
                else
                    nouvel_ensemble{k} = nouveau_vecteur;
                    ignorance          = [ignorance, nouveau_vecteur];
                    nouveau_vecteur    = [];
                    k                  = k + 1;
                end
            end
            nouvelle_info(g,h).elements = nouvel_ensemble;
            nouvelle_info(g,h).masses   = vecteur_masse(h);
            if isequal(g,1)
                matrice_monome(ligne,colonne).elements = nouvel_ensemble;
                matrice_monome(ligne,colonne).masses   = vecteur_masse(h);
                colonne                                = colonne + 1;
            elseif isequal(g,2)
                matrice_monome(ligne,colonne).elements = nouvel_ensemble;
                matrice_monome(ligne,colonne).masses   = vecteur_masse(h);
                ligne                                  = ligne + 1;
            end
        end
    end
    ignorance = unique(ignorance);
    for r = 1 : length(ignorance)
        ignorance_nouveau{r} = ignorance(r);
    end
    retour = matrice_monome;
%% case static
elseif isequal(sorte,2)
    %% variables
    f               = 1;
    %% go through each sources
    for g = 1 : n
        ensemble      = info(g).elements;
        vecteur_masse = info(g).masses;
        %% go through each set of elements
        for h = 1 : length(ensemble)
            vecteur         = ensemble{h};
            nouveau_vecteur = [];
            nouvel_ensemble = {};
            k               = 1;
             %% go through each element of the vector
            %%     to separate the products and sums
            for j = 1 : length(vecteur)
                if ~isequal(vecteur(j), ADD)
                    if ~isequal(nouveau_vecteur, []) & ~isequal(vecteur(j), MULT)
                        nouveau_vecteur = [nouveau_vecteur, vecteur(j)];
                        if isequal(j,length(vecteur))
                           nouvel_ensemble{k} = nouveau_vecteur;
                        end
                    elseif ~isequal(vecteur(j), MULT)
                        nouveau_vecteur = [vecteur(j)];
                        if isequal(j,length(vecteur))
                           nouvel_ensemble{k} = nouveau_vecteur;
                        end
                    end
                else
                    nouvel_ensemble{k} = nouveau_vecteur;
                    nouveau_vecteur    = [];
                    k                  = k + 1;
                end
            end
            ensemble_monome(f).elements = nouvel_ensemble;
            ensemble_monome(f).masses  = vecteur_masse(h);
            f                          = f + 1;
        end
    end
    ignorance = [];
    retour    = ensemble_monome;
 %% case contraint
elseif isequal(sorte,3)
    for g = 1 : length(info)
        vecteur         = info{g};
        nouveau_vecteur = [];
        nouvel_ensemble = {};
        k               = 1;
        for h = 1 : length(vecteur)
            if ~isequal(vecteur(h), ADD)
                 if ~isequal(nouveau_vecteur, []) & ~isequal(vecteur(h), MULT)
                     nouveau_vecteur = [nouveau_vecteur, vecteur(h)];
                     if isequal(h,length(vecteur))
                        nouvel_ensemble{k} = nouveau_vecteur;
                     end
                 elseif ~isequal(vecteur(h), MULT)
                     nouveau_vecteur = [vecteur(h)];
                     if isequal(h,length(vecteur))
                        nouvel_ensemble{k} = nouveau_vecteur;
                     end
                 end
            else
                 nouvel_ensemble{k} = nouveau_vecteur;
                 nouveau_vecteur    = [];
                 k                  = k + 1;
            end
        end
        nouvelle_contrainte{g} = nouvel_ensemble;
    end
    ignorance = [];
    retour    = nouvelle_contrainte;
end
```

15.7.19 File : separation_unique.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  separates products in input data, one info. at a time
%
% info:         information from sources (initial data)
% sorte:        type of separation
%
% matrice_monome:   separated data (products)
% ignorance:        total ignorance
% nombreElement:    number of vectors in sets of each information
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [matrice_monome,ignorance,nombreElement] = ...
                                        separation_unique(info,matrice_monome)
%#inbounds
%#realonly
global ADD
global MULT
global SOURCE
ADD    = -2;
MULT   = -1;
SOURCE = 2;
nouvelle_info  = []; %struc elements: set of vector
ignorance      = [];
if isequal(matrice_monome, [])
    ligne   = 1;
    colonne = 1;
else
    [m,n]   = size(matrice_monome);
    ligne   = m + 1;
    colonne = 1;
end
%% takes each elements of each sources and separates the products
[m,n] = size(info);
for g = 1 : n
    nombreElement(g) = length(info(g).elements);
end
%% go through each sources
for g = 1 : n
    ensemble      = info(g).elements;
    vecteur_masse = info(g).masses;
    %% go through each set of elements
    for h = 1 : length(ensemble)
        vecteur         = ensemble{h};
        nouveau_vecteur = [];
        nouvel_ensemble = {};
        k               = 1;
        %% go through each elements of the vector
        %% separates the products and sums
        for j = 1 : length(vecteur)
            if ~isequal(vecteur(j), ADD)
                if ~isequal(nouveau_vecteur, []) & ~isequal(vecteur(j), MULT)
                    nouveau_vecteur = [nouveau_vecteur, vecteur(j)];
                    if isequal(j,length(vecteur))
                       nouvel_ensemble{k} = nouveau_vecteur;
                       ignorance          = [ignorance, nouveau_vecteur];
                    end
                elseif ~isequal(vecteur(j), MULT)
                    nouveau_vecteur = [vecteur(j)];
                    if isequal(j,length(vecteur))
                       nouvel_ensemble{k} = nouveau_vecteur;
                       ignorance          = [ignorance, nouveau_vecteur];
                    end
                end
            else
                nouvel_ensemble{k} = nouveau_vecteur;
                ignorance          = [ignorance, nouveau_vecteur];
                nouveau_vecteur    = [];
                k                  = k + 1;
            end
        end
        nouvelle_info(g,h).elements           = nouvel_ensemble;
        nouvelle_info(g,h).masses             = vecteur_masse(h);
        matrice_monome(ligne,colonne).elements = nouvel_ensemble;
        matrice_monome(ligne,colonne).masses   = vecteur_masse(h);
        colonne                               = colonne + 1;
    end
end
ignorance = unique(ignorance);
```

15.7.20 File : somme_produit_complet.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description: function that converts input in sum of products
%
% matrice_contraire:     matrix in product of sums
%
% matrice_complet:     matrix in sum of products
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [matrice_complet] = somme_produit_complet(matrice_contraire);
%#inbounds
%#realonly
ensemble_elements = {};
vecteur           = [];
```

```
matrice           = [];
matrice_complet   = [];
p                 = 1;
ensembleN         = {};
[m,n]             = size(matrice_contraire);

if ~isempty(matrice_contraire(1,1).elements)
 u = 1;
 v = 1;
else
  u = 2;
  v = 2;
end
%% go through the sets and puts them in sum of product
for g = u : m
    for t = v : n
        ensemble_elements = matrice_contraire(g,t).elements;
        if ~isequal(ensemble_elements, {})
            matrice_complet(g,t).elements = {};
            matrice_complet(g,t).masses   = 0;
            ensembleN                     = {};
            if isequal(length(ensemble_elements), 1)
                vecteur      = ensemble_elements{1};
                j            = 1;
                ensembleN{j} = [];
                for f = 1 : length(vecteur)
                    ensembleN{j} = [vecteur(f)];
                    j            = j + 1;
                end
                matrice_complet(g,t).elements = ensembleN;
                matrice_complet(g,t).masses   = matrice_contraire(g,t).masses;
            elseif length(ensemble_elements) >= 2
                matrice_complet(g,t).elements = [];
                changer                       = 0;
                if length(ensemble_elements) >= 3
                    vecteur1 = ensemble_elements{1};
                    vecteur2 = ensemble_elements{2};
                    %file produit_somme_complet.m needed an '~' for the IF
                    %here to work as it should be.
                    if (length(vecteur1) > 1 & length(vecteur2) > 1)
                        matrice_complet(g,t).elements = ...
                                              matrice_contraire(g,t).elements;
                        matrice_complet(g,t).masses   = ...
                                              matrice_contraire(g,t).masses;
                    else
                        changer = 1 ;
                    end
                else
                    changer = 1;
                end
                  if isequal(changer,1);
                    matrice_complet(g,t).elements = {};
                    i                             = 1;
 for k = 1 : length(ensemble_elements) - 1
if (k < 2)
    if (k + 1) > length(ensemble_elements)
        x = length(ensemble_elements);
    else
        x = k + 1;
    end
     for w = k : x
        vecteur = ensemble_elements{w};
        j       = 1;
        for f = 1 : length(vecteur)
            if isequal(length(vecteur),1)
                ensembleN{j} = [vecteur];
            else
                ensembleN{j} = [vecteur(f)];
                j            = j  + 1;
            end
        end

        if isequal(i,1)
            matrice(1,2).elements = ensembleN;
            matrice(1,2).masses   = 0;
            ensembleN             = {};
            i                     = 2;
        elseif isequal(i,2)
            matrice(2,1).elements = ensembleN;
            matrice(2,1).masses   = 0;
            ensembleN             = {};
            i                     = 1;
         end
    end
elseif (k >= 2) & (length(ensemble_elements) > 2)
    w = k + 1;
    j = 1;
    vecteur = ensemble_elements{w};
    for f = 1 : length(vecteur)
        if isequal(length(vecteur),1)
           ensembleN{j} = [vecteur];
        else
           ensembleN{j} = [vecteur(f)];
           j            = j  + 1;
        end
    end
    matrice(1,2).elements = matrice_complet(g,t).elements;
    matrice(1,2).masses   = 0;

    matrice(2,1).elements = ensembleN;
    matrice(2,1).masses   = 0;
    ensembleN             = {};
end
      matrice              = ordre_grandeur(matrice,2);
    resultat             = union_matrice(matrice);
```

```
                        matrice(2,1).elements = {};
                        matrice(1,2).elements = {};

                        [s,b] = size(resultat);
                        for r = 1 : s
                        for d = 1 : b
                            masse = resultat(r,d).masses;
                            if isequal(masse, 0)
                                matrice_complet(g,t).elements = ...
                                                   resultat(r,d).elements;
                                matrice_complet(g,t).masses   = ...
                                                  matrice_contraire(g,t).masses;
                            end
                        end
                        end
                    end
                end
            elseif ~isequal(ensemble_elements, [])
                matrice_complet(g,t).elements = matrice_contraire(g,t).elements;
                matrice_complet(g,t).masses   = matrice_contraire(g,t).masses;
            end
        end
    end
end
if (g >= 2) & (t >= 2)
    matrice_complet = ordre_grandeur(matrice_complet,2);
end
```

15.7.21 File : tri.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  function that sorts the elements
%
% matrice:      matrix to sort, can be a set
% sorte:        type of input [matrix | set]
%
% retour:       matrix, or set, once the elements are sorted
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [retour] = tri(matrice,sorte)
%#inbounds
%#realonly
%% case matrix
if isequal(sorte,1)
    [m,n]         = size(matrice);
    ensemble_temp = [];
    if m > 1
        u = 2;
        v = 2;
    else
        u = 1;
        v = 1;
    end
    %% go through each elements of the matrix, sort them
    for h = u : m
        for g = v : n
            ensemble = matrice(h,g).elements;
            for f = 1 : length(ensemble)
                for k = f + 1 : length(ensemble)

                    %% if they are the same length, look at each number, at
                    %% order them
                    if isequal(length(ensemble{f}),length(ensemble{k}))
                        if (ensemble{f} > ensemble{k})
                            ensemble_temp = ensemble{f};
                            ensemble{f}   = ensemble{k};
                            ensemble{k}   = ensemble_temp;
                        end
                    else
                         %% ifnot the same length, put at first, the smaller
                        if length(ensemble{f}) > length(ensemble{k})
                            ensemble_temp = ensemble{f};
                            ensemble{f}   = ensemble{k};
                            ensemble{k}   = ensemble_temp;
                        end
                    end
                end
            end
            matriceOut(h,g).elements = ensemble;
            matriceOut(h,g).masses   = matrice(h,g).masses;
        end
    end
    retour = matriceOut;
%% case set
else
    ensemble_temp = [];
    %% go through each elements of the set, sort them
    for h = 1 : length(matrice)
        ensemble_tri = matrice{h};
        for f = 1 : length(ensemble_tri)
            for k = f + 1 : length(ensemble_tri)
                if isequal(length(ensemble_tri{f}),length(ensemble_tri{k}))
                    if (ensemble_tri{f} > ensemble_tri{k})
                        ensemble_temp   = ensemble_tri{f};
                        ensemble_tri{f} = ensemble_tri{k};
                        ensemble_tri{k} = ensemble_temp;
                    end
                else
                    if length(ensemble_tri{f}) > length(ensemble_tri{k})
                        ensemble_temp   = ensemble_tri{f};
                        ensemble_tri{f} = ensemble_tri{k};
                        ensemble_tri{k} = ensemble_temp;
                    end
                end
            end
        end
        ensembleOut{h} = ensemble_tri;
    end
    retour = ensembleOut;
end
```

15.7.22 File : union_matrice.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Description:  function that computes the union matrix and masses
%
% matrice_monome: objects and masses once separated, on the 1st line/column
%
% matrice_union : returns the result of the union and masses
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [matrice_union] = union_matrice(matrice_monome)
%#inbounds
%#realonly
matrice_union = [];
[m,n]         = size(matrice_monome);
ensembleN     = {};
ensembleM     = {};
vecteurN      = [];
vecteurM      = [];
ensemble1     = [];
ensemble2     = [];
%% go through the 1st line and column, fill the union matrix
for g = 2 : n
    ensembleN = matrice_monome(1,g).elements;
    if ~isequal(ensembleN,{})
        for h = 2 : m
            ensembleM = matrice_monome(h,1).elements;
            if ~isequal(ensembleM,{})
                if isequal(ensembleN, ensembleM)
                    matrice_union(h,g).elements = ensembleN;
                    matrice_union(h,g).masses   = matrice_monome(1,g).masses *...
                                                  matrice_monome(h,1).masses;
                else
                     %% put the identical ones (from same line) togheter
                    [ensembleM,ensembleN] = bon_ordre(ensembleM,ensembleN);
                    %% verifies which one is the higher
                    if length(ensembleM) >= length(ensembleN)
                        ensemble1 = ensembleN;
                        ensemble2 = ensembleM;
                    else
                        ensemble1 = ensembleM;
                        ensemble2 = ensembleN;
                    end
                end
                 %% fill the union matrix
                nouvel_ensemble = {};
                j               = 1;
                for t = 1 : length(ensemble1)
                    for s = 1 : length(ensemble2)
                        if t <= length(ensemble2)
```

```
                            if isequal(ensemble2{s},ensemble1{t})
                                nouvel_ensemble{j} = [ensemble1{t}];
                            else
                                vecteur = [ensemble2{s},ensemble1{t}];
                                nouvel_ensemble{j} = unique(vecteur);
                            end
                        else
                            if isequal(ensemble1{length(ensemble2)},ensemble1{t})
                                nouvel_ensemble{j} = [ensemble1{t}];
                            else
                                vecteur = ...
                                     [ensemble1{length(ensemble2)},ensemble1{t}];
                                nouvel_ensemble{j} = unique(vecteur);
                            end
                        end
                        j = j + 1;
                    end
                end
                matrice_union(h,g).elements = nouvel_ensemble;
                matrice_union(h,g).masses   = matrice_monome(1,g).masses *...
                                             matrice_monome(h,1).masses;
            end
        end
    end
end
matrice_union = ordre_grandeur(matrice_union,2);
```