

The FMB algorithm  
An intersection detection algorithm for 2D/3D  
cuboid and tetrahedron based on the  
Fourier-Motzkin elimination method

P. Baillehache

November 24, 2019

**Abstract**

In this paper I introduce how to perform intersection detection of pair of cuboid/tetrahedron in 2D and 3D by using the Fourier-Motzkin elimination method. The mathematical definition and solution of the problem in the two first sections is followed by the algorithm of the solution and its implementation in the C programming language in the two following sections. The last two sections introduce the validation and qualification of the FMB algorithm against the SAT algorithm.

# Contents

<b>1</b>	<b>Notations</b>	<b>3</b>
<b>2</b>	<b>Definition of the problem</b>	<b>3</b>
2.1	Static case . . . . .	3
2.2	Dynamic case . . . . .	6
<b>3</b>	<b>Solution</b>	<b>10</b>
3.1	Fourier-Motzkin elimination method . . . . .	10
3.2	Application of the Fourier-Motzkin method to the intersection problem . . . . .	11
<b>4</b>	<b>Algorithms</b>	<b>12</b>
4.1	2D case . . . . .	12
4.2	3D case . . . . .	12
<b>5</b>	<b>Implementation</b>	<b>12</b>
5.1	Frames . . . . .	13
5.1.1	Header . . . . .	13
5.1.2	Body . . . . .	14
5.2	FMB . . . . .	25
5.2.1	2D static . . . . .	25
5.2.2	3D static . . . . .	37
5.2.3	2D dynamic . . . . .	51
5.2.4	3D dynamic . . . . .	63
5.3	Example of use . . . . .	77
<b>6</b>	<b>Validation</b>	<b>79</b>
6.1	Code . . . . .	79
6.1.1	Unit tests . . . . .	79
6.1.2	Validation against SAT . . . . .	98
6.2	Results . . . . .	104
6.2.1	2D . . . . .	104
6.2.2	3D . . . . .	104
<b>7</b>	<b>Qualification</b>	<b>104</b>
7.1	Code . . . . .	104
7.2	Results . . . . .	116
7.2.1	2D . . . . .	116
7.2.2	3D . . . . .	117

<b>8</b>	<b>Conclusion</b>	<b>118</b>
<b>9</b>	<b>Annex</b>	<b>119</b>
9.1	SAT implementation . . . . .	119
9.1.1	Header . . . . .	119
9.1.2	Body . . . . .	119
9.2	Makefile . . . . .	128

## 1 Notations

- $[M]_{r,c}$  is the component at column  $c$  and row  $r$  of the matrix  $M$
- $[V]_r$  is the  $r$ -th component of the vector  $\vec{V}$

## 2 Definition of the problem

### 2.1 Static case

In this paper I'll use the term "Frame" to speak indifferently of cuboid and tetrahedron.

The two Frames are represented as a vector origin and a number of component vectors equal to the dimension  $D$  of the space where live the Frames. Each vector is of dimension equal to  $D$ .

Lets call  $\mathbb{A}$  and  $\mathbb{B}$  the two Frames tested for intersection. If  $A$  and  $B$  are two cuboids:

$$\mathbb{A} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ \vec{O}_{\mathbb{A}} + C_{\mathbb{A}} \cdot \vec{X} \end{array} \right\} \quad (1)$$

$$\mathbb{B} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ \vec{O}_{\mathbb{B}} + C_{\mathbb{B}} \cdot \vec{X} \end{array} \right\} \quad (2)$$

where  $\vec{O}_{\mathbb{A}}$  is the origin of  $\mathbb{A}$  and  $C_{\mathbb{A}}$  is the matrix of the components of  $A$  (one component per column). Idem for  $\vec{O}_{\mathbb{B}}$  and  $C_{\mathbb{B}}$ .

If  $\mathbb{A}$  and  $\mathbb{B}$  are two tetrahedrons:

$$\mathbb{A} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ \vec{O}_{\mathbb{A}} + C_{\mathbb{A}} \cdot \vec{X} \end{array} \right\} \quad (3)$$

$$\mathbb{B} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ \vec{O}_{\mathbb{B}} + C_{\mathbb{B}} \cdot \vec{X} \end{array} \right\} \quad (4)$$

I'll assume the Frames are well formed, i.e. their components matrix is invertible. It is then possible to express  $\mathbb{B}$  in  $\mathbb{A}$ 's coordinates system, noted as  $\mathbb{B}_{\mathbb{A}}$ . If  $\mathbb{B}$  is a cuboid:

$$\mathbb{B}_{\mathbb{A}} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X}) \end{array} \right\} \quad (5)$$

If  $\mathbb{B}$  is a tetrahedron:

$$\mathbb{B}_{\mathbb{A}} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X}) \end{array} \right\} \quad (6)$$

$\mathbb{A}$  in its own coordinates system becomes, for a cuboid:

$$\mathbb{A}_{\mathbb{A}} = \left\{ \vec{X} \in [0.0, 1.0]^D \right\} \quad (7)$$

and for a tetrahedron:

$$\mathbb{A}_{\mathbb{A}} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \end{array} \right\} \quad (8)$$

The intersection of  $\mathbb{A}$  and  $\mathbb{B}$  in  $\mathbb{A}$ 's coordinates sytem, can then be expressed as follow.

If  $\mathbb{A}$  and  $\mathbb{B}$  are two cuboids:

$$\left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X}) \cap [0.0, 1.0]^D \end{array} \right\} \quad (9)$$

If  $\mathbb{A}$  is a cuboid and  $\mathbb{B}$  is a tetrahedron:

$$\left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X}) \cap [0.0, 1.0]^D \end{array} \right\} \quad (10)$$

If  $\mathbb{A}$  is a tetrahedron and  $\mathbb{B}$  is a cuboid:

$$\left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X}) \cap [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X})]_i \leq 1.0 \end{array} \right\} \quad (11)$$

If  $\mathbb{A}$  and  $\mathbb{B}$  are two tetrahedrons:

$$\left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X}) \cap [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} \left[ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X}) \right]_i \leq 1.0 \end{array} \right\} \quad (12)$$

These can in turn be expressed as systems of linear inequations as follows, given the two shortcuts  $\vec{O}_{\mathbb{B}_{\mathbb{A}}} = C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}})$  and  $C_{\mathbb{B}_{\mathbb{A}}} = C_{\mathbb{A}}^{-1} \cdot C_{\mathbb{B}}$ .

If  $\mathbb{A}$  and  $\mathbb{B}$  are two cuboids:

$$\left\{ \begin{array}{ll} [X]_0 & \leq 1.0 \\ -[X]_0 & \leq 0.0 \\ \dots & \\ [X]_{D-1} & \leq 1.0 \\ -[X]_{D-1} & \leq 0.0 \\ \sum_{i=0}^{D-1} [C_{\mathbb{B}_{\mathbb{A}}}]_{0,i} \cdot [X]_i & \leq 1.0 - [O_{\mathbb{B}_{\mathbb{A}}}]_0 \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_{\mathbb{A}}}]_{0,i} \cdot [X]_i & \leq -[O_{\mathbb{B}_{\mathbb{A}}}]_0 \\ \dots & \\ \sum_{i=0}^{D-1} [C_{\mathbb{B}_{\mathbb{A}}}]_{D-1,i} \cdot [X]_i & \leq 1.0 - [O_{\mathbb{B}_{\mathbb{A}}}]_{D-1} \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_{\mathbb{A}}}]_{D-1,i} \cdot [X]_i & \leq -[O_{\mathbb{B}_{\mathbb{A}}}]_{D-1} \end{array} \right. \quad (13)$$

If  $\mathbb{A}$  is a cuboid and  $\mathbb{B}$  is a tetrahedron:

$$\left\{ \begin{array}{ll} -[X]_0 & \leq 0.0 \\ \dots & \\ -[X]_{D-1} & \leq 0.0 \\ \sum_{i=0}^{D-1} [C_{\mathbb{B}_{\mathbb{A}}}]_{0,i} \cdot [X]_i & \leq 1.0 - [O_{\mathbb{B}_{\mathbb{A}}}]_0 \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_{\mathbb{A}}}]_{0,i} \cdot [X]_i & \leq -[O_{\mathbb{B}_{\mathbb{A}}}]_0 \\ \dots & \\ \sum_{i=0}^{D-1} [C_{\mathbb{B}_{\mathbb{A}}}]_{D-1,i} \cdot [X]_i & \leq 1.0 - [O_{\mathbb{B}_{\mathbb{A}}}]_{D-1} \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_{\mathbb{A}}}]_{D-1,i} \cdot [X]_i & \leq -[O_{\mathbb{B}_{\mathbb{A}}}]_{D-1} \\ \sum_{i=0}^{D-1} [X]_i & \leq 1.0 \end{array} \right. \quad (14)$$

If  $\mathbb{A}$  is a tetrahedron and  $\mathbb{B}$  is a cuboid:

$$\left\{ \begin{array}{l} [X]_0 \leq 1.0 \\ -[X]_0 \leq 0.0 \\ \dots \\ [X]_{D-1} \leq 1.0 \\ -[X]_{D-1} \leq 0.0 \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{0,i} \cdot [X]_i \leq -[O_{\mathbb{B}_A}]_0 \\ \dots \\ \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} \cdot [X]_i \leq 1.0 - [O_{\mathbb{B}_A}]_{D-1} \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} \cdot [X]_i \leq -[O_{\mathbb{B}_A}]_{D-1} \\ \sum_{j=0}^{D-1} \left( \left( \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{j,i} \right) \cdot [X]_i \right) \leq 1.0 - \sum_{i=0}^{D-1} [O_{\mathbb{B}_A}]_i \end{array} \right. \quad (15)$$

If  $\mathbb{A}$  and  $\mathbb{B}$  are two tetrahedrons:

$$\left\{ \begin{array}{l} -[X]_0 \leq 0.0 \\ \dots \\ -[X]_{D-1} \leq 0.0 \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{0,i} \cdot [X]_i \leq -[O_{\mathbb{B}_A}]_0 \\ \dots \\ \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} \cdot [X]_i \leq 1.0 - [O_{\mathbb{B}_A}]_{D-1} \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} \cdot [X]_i \leq -[O_{\mathbb{B}_A}]_{D-1} \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ \sum_{j=0}^{D-1} \left( \left( \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{j,i} \right) \cdot [X]_i \right) \leq 1.0 - \sum_{i=0}^{D-1} [O_{\mathbb{B}_A}]_i \end{array} \right. \quad (16)$$

## 2.2 Dynamic case

If the frames  $\mathbb{A}$  and  $\mathbb{B}$  are moving linearly along the vectors  $\vec{V}_A$  and  $\vec{V}_B$  respectively during the interval of time  $t \in [0.0, 1.0]$ , the above definition of the problem is modified as follow.

If  $A$  and  $B$  are two cuboids:

$$\mathbb{A} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \vec{O}_A + C_A \cdot \vec{X} + \vec{V}_A \cdot t \end{array} \right\} \quad (17)$$

$$\mathbb{B} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \vec{O}_B + C_B \cdot \vec{X} + \vec{V}_B \cdot t \end{array} \right\} \quad (18)$$

where  $\vec{O}_\mathbb{A}$  is the origin of  $\mathbb{A}$  and  $C_\mathbb{A}$  is the matrix of the components of  $A$  (one component per column). Idem for  $\vec{O}_\mathbb{B}$  and  $C_\mathbb{B}$ .

If  $\mathbb{A}$  and  $\mathbb{B}$  are two tetrahedrons:

$$\mathbb{A} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ \vec{O}_\mathbb{A} + C_\mathbb{A} \cdot \vec{X} + \vec{V}_\mathbb{A} \cdot t \end{array} \right\} \quad (19)$$

$$\mathbb{B} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ \vec{O}_\mathbb{B} + C_\mathbb{B} \cdot \vec{X} + \vec{V}_\mathbb{B} \cdot t \end{array} \right\} \quad (20)$$

If  $\mathbb{B}$  is a cuboid,  $\mathbb{B}_\mathbb{A}$  becomes:

$$\mathbb{B}_\mathbb{A} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ C_\mathbb{A}^{-1} \cdot (\vec{O}_\mathbb{B} - \vec{O}_\mathbb{A} + C_\mathbb{B} \cdot \vec{X} + (\vec{V}_\mathbb{B} - \vec{V}_\mathbb{A}) \cdot t) \end{array} \right\} \quad (21)$$

If  $\mathbb{B}$  is a tetrahedron:

$$\mathbb{B}_\mathbb{A} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_\mathbb{A}^{-1} \cdot (\vec{O}_\mathbb{B} - \vec{O}_\mathbb{A} + C_\mathbb{B} \cdot \vec{X} + (\vec{V}_\mathbb{B} - \vec{V}_\mathbb{A}) \cdot t) \end{array} \right\} \quad (22)$$

$\mathbb{A}$  in its own coordinates system has the same definition as in the static case. For a cuboid:

$$\mathbb{A}_\mathbb{A} = \left\{ \vec{X} \in [0.0, 1.0]^D \right\} \quad (23)$$

and for a tetrahedron:

$$\mathbb{A}_\mathbb{A} = \left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \end{array} \right\} \quad (24)$$

The intersection of  $\mathbb{A}$  and  $\mathbb{B}$  in  $\mathbb{A}$ 's coordinates sytem, can then be expressed as follow.

If  $\mathbb{A}$  and  $\mathbb{B}$  are two cuboids:

$$\left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X} + (\vec{V}_{\mathbb{B}} - \vec{V}_{\mathbb{A}}) \cdot t) \cap [0.0, 1.0]^D \end{array} \right\} \quad (25)$$

If  $\mathbb{A}$  is a cuboid and  $\mathbb{B}$  is a tetrahedron:

$$\left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X} + (\vec{V}_{\mathbb{B}} - \vec{V}_{\mathbb{A}}) \cdot t) \cap [0.0, 1.0]^D \end{array} \right\} \quad (26)$$

If  $\mathbb{A}$  is a tetrahedron and  $\mathbb{B}$  is a cuboid:

$$\left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X} + (\vec{V}_{\mathbb{B}} - \vec{V}_{\mathbb{A}}) \cdot t) \cap [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X})]_i \leq 1.0 \end{array} \right\} \quad (27)$$

If  $\mathbb{A}$  and  $\mathbb{B}$  are two tetrahedrons:

$$\left\{ \begin{array}{l} \vec{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X} + (\vec{V}_{\mathbb{B}} - \vec{V}_{\mathbb{A}}) \cdot t) \cap [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}} + C_{\mathbb{B}} \cdot \vec{X})]_i \leq 1.0 \end{array} \right\} \quad (28)$$

These lead to the systems of linear inequations as follows, given the three shortcuts  $\vec{O}_{\mathbb{B}\mathbb{A}} = C_{\mathbb{A}}^{-1} \cdot (\vec{O}_{\mathbb{B}} - \vec{O}_{\mathbb{A}})$ ,  $\vec{V}_{\mathbb{B}\mathbb{A}} = C_{\mathbb{A}}^{-1} \cdot (\vec{V}_{\mathbb{B}} - \vec{V}_{\mathbb{A}})$  and  $C_{\mathbb{B}\mathbb{A}} = C_{\mathbb{A}}^{-1} \cdot C_{\mathbb{B}}$ .



If  $\mathbb{A}$  and  $\mathbb{B}$  are two cuboids:

$$\left\{ \begin{array}{rcl} t & \leq & 1.0 \\ -t & \leq & 0.0 \\ [X]_0 & \leq & 1.0 \\ -[X]_0 & \leq & 0.0 \\ \dots & & \\ [X]_{D-1} & \leq & 1.0 \\ -[X]_{D-1} & \leq & 0.0 \\ [V_{\mathbb{B}_A}]_0 t + \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{0,i} [X]_i & \leq & 1.0 - [O_{\mathbb{B}_A}]_0 \\ -[V_{\mathbb{B}_A}]_0 t - \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{0,i} [X]_i & \leq & -[O_{\mathbb{B}_A}]_0 \\ \dots & & \\ [V_{\mathbb{B}_A}]_{D-1} t + \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} [X]_i & \leq & 1.0 - [O_{\mathbb{B}_A}]_{D-1} \\ -[V_{\mathbb{B}_A}]_{D-1} t - \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} [X]_i & \leq & -[O_{\mathbb{B}_A}]_{D-1} \end{array} \right. \quad (29)$$

If  $\mathbb{A}$  is a cuboid and  $\mathbb{B}$  is a tetrahedron:

$$\left\{ \begin{array}{rcl} t & \leq & 1.0 \\ -t & \leq & 0.0 \\ -[X]_0 & \leq & 0.0 \\ \dots & & \\ -[X]_{D-1} & \leq & 0.0 \\ \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{0,i} [X]_i & \leq & 1.0 - [O_{\mathbb{B}_A}]_0 \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{0,i} [X]_i & \leq & -[O_{\mathbb{B}_A}]_0 \\ \dots & & \\ [V_{\mathbb{B}_A}]_{D-1} t + \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} [X]_i & \leq & 1.0 - [O_{\mathbb{B}_A}]_{D-1} \\ -[V_{\mathbb{B}_A}]_{D-1} t - \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} [X]_i & \leq & -[O_{\mathbb{B}_A}]_{D-1} \\ \sum_{i=0}^{D-1} [X]_i & \leq & 1.0 \end{array} \right. \quad (30)$$

If  $\mathbb{A}$  is a tetrahedron and  $\mathbb{B}$  is a cuboid:

$$\left\{ \begin{array}{rcl} t & \leq & 1.0 \\ -t & \leq & 0.0 \\ [X]_0 & \leq & 1.0 \\ -[X]_0 & \leq & 0.0 \\ \dots & & \\ [X]_{D-1} & \leq & 1.0 \\ -[X]_{D-1} & \leq & 0.0 \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{0,i} [X]_i & \leq & -[O_{\mathbb{B}_A}]_0 \\ \dots & & \\ [V_{\mathbb{B}_A}]_{D-1} t + \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} [X]_i & \leq & 1.0 - [O_{\mathbb{B}_A}]_{D-1} \\ -[V_{\mathbb{B}_A}]_{D-1} t - \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} [X]_i & \leq & -[O_{\mathbb{B}_A}]_{D-1} \\ \sum_{j=0}^{D-1} \left( \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{j,i} [X]_i \right) & \leq & 1.0 - \sum_{i=0}^{D-1} [O_{\mathbb{B}_A}]_i \end{array} \right. \quad (31)$$

If  $\mathbb{A}$  and  $\mathbb{B}$  are two tetrahedrons:

$$\left\{ \begin{array}{rcl} t & \leq & 1.0 \\ -t & \leq & 0.0 \\ -[X]_0 & \leq & 0.0 \\ \dots & & \\ -[X]_{D-1} & \leq & 0.0 \\ -\sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{0,i} [X]_i & \leq & -[O_{\mathbb{B}_A}]_0 \\ \dots & & \\ [V_{\mathbb{B}_A}]_{D-1} t + \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} [X]_i & \leq & 1.0 - [O_{\mathbb{B}_A}]_{D-1} \\ -[V_{\mathbb{B}_A}]_{D-1} t - \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{D-1,i} [X]_i & \leq & -[O_{\mathbb{B}_A}]_{D-1} \\ \sum_{i=0}^{D-1} [X]_i & \leq & 1.0 \\ \sum_{j=0}^{D-1} \left( \left( \sum_{i=0}^{D-1} [C_{\mathbb{B}_A}]_{j,i} \right) [X]_i \right) & \leq & 1.0 - \sum_{i=0}^{D-1} [O_{\mathbb{B}_A}]_i \end{array} \right. \quad (32)$$

### 3 Solution

#### 3.1 Fourier-Motzkin elimination method

The Fourier-Motzkin elimination method has been introduced by J.J.-B. Fourier in 1827 [1], and described in the Ph.D. thesis of T.S. Motzkin in 1936 [2]. This is a generalization of the Gaussian elimination method to linear systems of inequalities. This method consists of eliminating one variable of the system and rewrite a new system accordingly. Then the elimination operation is repeated on another variable in the new system, and so on until we obtain a trivial system with only one variable. From there, a solution for each variable can be obtained if it exists. The variable elimination is performed as follow.

Lets write the linear system  $\mathcal{I}$  of  $m$  inequalities and  $n$  variables as

$$\left\{ \begin{array}{rcl} a_{11}.x_1 & +a_{12}.x_2 & +\dots +a_{1n}.x_n \leq b_1 \\ a_{21}.x_1 & +a_{22}.x_2 & +\dots +a_{2n}.x_n \leq b_2 \\ & & \vdots \\ a_{m1}.x_1 & +a_{m2}.x_2 & +\dots +a_{mn}.x_n \leq b_m \end{array} \right. \quad (33)$$

with

$$\begin{aligned} i & \in 1, 2, \dots, m \\ j & \in 1, 2, \dots, n \\ x_i & \in \mathbb{R} \\ a_{ij} & \in \mathbb{R} \\ b_j & \in \mathbb{R} \end{aligned} \quad (34)$$

To eliminate the first variable  $x_1$ , let's multiply each inequality by  $1.0/|a_{i1}|$  where  $a_{i1} \neq 0.0$ . The system becomes

$$\begin{cases} x_1 + a'_{i2}.x_2 + \dots + a'_{in}.x_n \leq b'_i & (i \in \mathcal{I}_+) \\ a_{i2}.x_2 + \dots + a_{in}.x_n \leq b_i & (i \in \mathcal{I}_0) \\ -x_1 + a'_{i2}.x_2 + \dots + a'_{in}.x_n \leq b'_i & (i \in \mathcal{I}_-) \end{cases} \quad (35)$$

where

$$\mathcal{I}_+ = \{i : a_{i1} > 0.0\}$$

$$\mathcal{I}_0 = \{i : a_{i1} = 0.0\}$$

$$\mathcal{I}_- = \{i : a_{i1} < 0.0\}$$

$$a'_{ij} = a_{ij}/|a_{i1}|$$

$$b'_i = b_i/|a_{i1}|$$

Then  $x_1, x_2, \dots, x_n \in \mathbb{R}^n$  is a solution of  $\mathcal{I}$  if and only if

$$\begin{cases} \sum_{j=2}^n ((a'_{kj} + a'_{lj}).x_j) \leq b'_k + b'_l & (k \in \mathcal{I}_+, l \in \mathcal{I}_-) \\ \sum_{j=2}^n (a_{ij}.x_j) \leq b_i & i \in \mathcal{I}_0 \end{cases} \quad (36)$$

and

$$\max_{l \in \mathcal{I}_-} \left( \sum_{j=2}^n (a'_{lj}.x_j) - b'_l \right) \leq x_1 \leq \min_{k \in \mathcal{I}_+} \left( b'_k - \sum_{j=2}^n (a'_{kj}.x_j) \right) \quad (37)$$

The same method is then applied on this new system to eliminate the second variable  $x_2$ , and so on until we reach the inequality

$$\max_{l \in \mathcal{I}_-'' \dots'} (-b_l'' \dots') \leq x_n \leq \min_{k \in \mathcal{I}_+'' \dots'} (b_k'' \dots') \quad (38)$$

If this inequality has no solution, then neither the system  $\mathcal{I}$ . If it has a solution, the minimum and maximum are the bounding values for the variable  $x_n$ . One can get a particular solution to the system  $\mathcal{I}$  by choosing a value for  $x_n$  between these bounding values, which allow us to set a particular value for the variable  $x_{n-1}$ , and so on back up to  $x_1$ .

### 3.2 Application of the Fourier-Motzkin method to the intersection problem

The Fourier-Motzkin method can be directly applied to obtain the bounds of each variable, if the system has a solution. If the system has no solution, the method will eventually reach an inconsistent inequality.

One solution  $\vec{S}$  within the bounds obtained by the resolution of the system is expressed in the Frame  $\mathbb{B}$ 's coordinates system. One can get the equivalent coordinates  $\vec{S}'$  in the real world's coordinates system as follow:

$$\vec{S}' = \vec{O}_{\mathbb{B}} + C_{\mathbb{B}} \cdot \vec{S} \quad (39)$$

Only one inconsistent inequality is sufficient to prove the absence of solution, and then the non intersection of the Frames. One shall check the inconsistency of each inequality as soon as possible during the resolution of the system to optimize the speed of the algorithm.

A sufficient condition for one inequality  $\sum_i a_i X_i \leq Y$  to be inconsistent is, given that  $\forall i, X_i \in [0.0, 1.0]$ :

$$Y < \sum_{i \in I^-} a_i \quad (40)$$

where  $I^- = \{i, a_i < 0.0\}$ .

## 4 Algorithms

In this section I introduce the algorithms of the solution of the previous section for the cases 2D and 3D.

### 4.1 2D case

algo

### 4.2 3D case

algo

## 5 Implementation

In this section I introduce an implementation of the algorithms of the previous section in the C language.

## 5.1 Frames

### 5.1.1 Header

```
#ifndef __FRAME_H_
#define __FRAME_H_

// ----- Includes -----

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

// ----- Macros -----

// ----- Enumerations -----

typedef enum {
    FrameCuboid,
    FrameTetrahedron
} FrameType;

// ----- Data structures -----

// Axis aligned bounding box structure
typedef struct {
    double min[2];
    double max[2];
} AABB2D;

typedef struct {
    double min[3];
    double max[3];
} AABB3D;

// Axis unaligned parallelepiped and tetrahedron structure
typedef struct {
    FrameType type;
    double orig[2];
    double speed[2];
    double comp[2][2];
    // AABB of the frame
    AABB2D bdgBox;
    // Inverted components used during computation
    double invComp[2][2];
} Frame2D;

typedef struct {
    FrameType type;
    double orig[3];
    double speed[3];
    double comp[3][3];
    // AABB of the frame
    AABB3D bdgBox;
    // Inverted components used during computation
    double invComp[3][3];
} Frame3D;

// ----- Functions declaration -----

// Print the AABB 'that' on stdout
```

```

// Output format is (min[0], min[1], min[2])-(max[0], max[1], max[2])
void AABBB2DPrint(const AABBB2D* const that);
void AABBB3DPrint(const AABBB3D* const that);

// Print the Frame 'that' on stdout
// Output format is (orig[0], orig[1], orig[2])
// (comp[0][0], comp[0][1], comp[0][2])
// (comp[1][0], comp[1][1], comp[1][2])
// (comp[2][0], comp[2][1], comp[2][2])
void Frame2DPrint(const Frame2D* const that);
void Frame3DPrint(const Frame3D* const that);

// Create a static Frame structure of FrameType 'type',
// at position 'orig' with components 'comp' ([iComp][iAxis])
Frame2D Frame2DCreateStatic(
    const FrameType type,
    const double orig[2],
    const double comp[2][2]);
Frame3D Frame3DCreateStatic(
    const FrameType type,
    const double orig[3],
    const double comp[3][3]);

// Project the Frame 'Q' in the Frame 'P' 's coordinates system and
// memorize the result in the Frame 'Qp'
void Frame2DImportFrame(
    const Frame2D* const P,
    const Frame2D* const Q,
    Frame2D* const Qp);
void Frame3DImportFrame(
    const Frame3D* const P,
    const Frame3D* const Q,
    Frame3D* const Qp);

// Export the AABBB 'bdgBox' from 'that' 's coordinates system to
// the real coordinates system and update 'bdgBox' with the resulting
// AABBB
void Frame2DExportBdgBox(
    const Frame2D* const that,
    const AABBB2D* const bdgBox,
    AABBB2D* const bdgBoxProj);
void Frame3DExportBdgBox(
    const Frame3D* const that,
    const AABBB3D* const bdgBox,
    AABBB3D* const bdgBoxProj);

// Power function for integer base and exponent
// Return 'base' ^ 'exp'
int powi(
    int base,
    unsigned int exp);

#endif

```

## 5.1.2 Body

```

#include "frame.h"

// ----- Macros -----

#define EPSILON 0.0000001

```

```

// ----- Functions declaration -----

// Update the inverse components of the Frame 'that'
void Frame2DUpdateInv(Frame2D* const that);
void Frame3DUpdateInv(Frame3D* const that);

// ----- Functions implementation -----

// Create a static Frame structure of FrameType 'type',
// at position 'orig' with components 'comp'
// arrangement is comp[iComp][iAxis]
Frame2D Frame2DCreateStatic(
    const FrameType type,
    const double orig[2],
    const double comp[2][2]) {

    // Create the new Frame
    Frame2D that;
    that.type = type;
    for (int iAxis = 2;
        iAxis--;) {

        that.orig[iAxis] = orig[iAxis];

        for (int iComp = 2;
            iComp--;) {

            that.comp[iComp][iAxis] = comp[iComp][iAxis];

        }
    }

    // Create the bounding box
    for (int iAxis = 2;
        iAxis--;) {

        double min = orig[iAxis];
        double max = orig[iAxis];

        for (int iComp = 2;
            iComp--;) {

            if (that.type == FrameCuboid) {

                if (that.comp[iComp][iAxis] < 0.0) {

                    min += that.comp[iComp][iAxis];

                }

                if (that.comp[iComp][iAxis] > 0.0) {

                    max += that.comp[iComp][iAxis];

                }

            } else if (that.type == FrameTetrahedron) {

                if (that.comp[iComp][iAxis] < 0.0 &&
                    min > orig[iAxis] + that.comp[iComp][iAxis]) {


```

```

        min = orig[iAxis] + that.comp[iComp][iAxis];
    }

    if (that.comp[iComp][iAxis] > 0.0 &&
        max < orig[iAxis] + that.comp[iComp][iAxis]) {

        max = orig[iAxis] + that.comp[iComp][iAxis];
    }
}

that.bdgBox.min[iAxis] = min;
that.bdgBox.max[iAxis] = max;
}

// Calculate the inverse matrix
Frame2DUpdateInv(&that);

// Return the new Frame
return that;
}

Frame3D Frame3DCreateStatic(
    const FrameType type,
    const double orig[3],
    const double comp[3][3]) {

    // Create the new Frame
    Frame3D that;
    that.type = type;
    for (int iAxis = 3;
        iAxis--;) {

        that.orig[iAxis] = orig[iAxis];

        for (int iComp = 3;
            iComp--;) {

            that.comp[iComp][iAxis] = comp[iComp][iAxis];
        }
    }

    // Create the bounding box
    for (int iAxis = 3;
        iAxis--;) {

        double min = orig[iAxis];
        double max = orig[iAxis];

        for (int iComp = 3;
            iComp--;) {

```



```

    if (that.type == FrameCuboid) {

        if (that.comp[iComp][iAxis] < 0.0) {

            min += that.comp[iComp][iAxis];

        }

        if (that.comp[iComp][iAxis] > 0.0) {

            max += that.comp[iComp][iAxis];

        }

    } else if (that.type == FrameTetrahedron) {

        if (that.comp[iComp][iAxis] < 0.0 &&
            min > orig[iAxis] + that.comp[iComp][iAxis]) {

            min = orig[iAxis] + that.comp[iComp][iAxis];

        }

        if (that.comp[iComp][iAxis] > 0.0 &&
            max < orig[iAxis] + that.comp[iComp][iAxis]) {

            max = orig[iAxis] + that.comp[iComp][iAxis];

        }

    }

}

that.bdgBox.min[iAxis] = min;
that.bdgBox.max[iAxis] = max;

}

// Calculate the inverse matrix
Frame3DUpdateInv(&that);

// Return the new Frame
return that;

}

// Update the inverse components of the Frame 'that'
void Frame2DUpdateInv(Frame2D* const that) {

    // Shortcuts
    double (*tc)[2] = that->comp;
    double (*tic)[2] = that->invComp;

    double det = tc[0][0] * tc[1][1] - tc[1][0] * tc[0][1];
    if (fabs(det) < EPSILON) {
        fprintf(stderr,
            "FrameUpdateInv: det == 0.0\n");
        exit(1);
    }
}

```

```

tic[0][0] = tc[1][1] / det;
tic[0][1] = -1.0 * tc[0][1] / det;
tic[1][0] = -1.0 * tc[1][0] / det;
tic[1][1] = tc[0][0] / det;
}

void Frame3DUpdateInv(Frame3D* const that) {

    // Shortcuts
    double (*tc)[3] = that->comp;
    double (*tic)[3] = that->invComp;

    // Update the inverse components
    double det =
        tc[0][0] * (tc[1][1] * tc[2][2] - tc[1][2] * tc[2][1]) -
        tc[1][0] * (tc[0][1] * tc[2][2] - tc[0][2] * tc[2][1]) +
        tc[2][0] * (tc[0][1] * tc[1][2] - tc[0][2] * tc[1][1]);
    if (fabs(det) < EPSILON) {
        fprintf(stderr,
            "FrameUpdateInv: det == 0.0\n");
        exit(1);
    }

    tic[0][0] = (tc[1][1]*tc[2][2]-tc[2][1]*tc[1][2]) / det;
    tic[0][1] = (tc[2][1]*tc[0][2]-tc[2][2]*tc[0][1]) / det;
    tic[0][2] = (tc[0][1]*tc[1][2]-tc[0][2]*tc[1][1]) / det;
    tic[1][0] = (tc[2][0]*tc[1][2]-tc[2][2]*tc[1][0]) / det;
    tic[1][1] = (tc[0][0]*tc[2][2]-tc[2][0]*tc[0][2]) / det;
    tic[1][2] = (tc[0][2]*tc[1][0]-tc[1][2]*tc[0][0]) / det;
    tic[2][0] = (tc[1][0]*tc[2][1]-tc[2][0]*tc[1][1]) / det;
    tic[2][1] = (tc[0][1]*tc[2][0]-tc[2][1]*tc[0][0]) / det;
    tic[2][2] = (tc[0][0]*tc[1][1]-tc[1][0]*tc[0][1]) / det;
}

// Project the Frame 'Q' in the Frame 'P' 's coordinates system and
// memorize the result in the Frame 'Qp'
void Frame2DImportFrame(
    const Frame2D* const P,
    const Frame2D* const Q,
    Frame2D* const Qp) {

    // Shortcuts
    const double* qo = Q->orig;
    double* qpo = Qp->orig;
    const double* po = P->orig;

    const double (*pi)[2] = P->invComp;
    double (*qpc)[2] = Qp->comp;
    const double (*qc)[2] = Q->comp;

    // Calculate the projection
    double v[2];
    for (int i = 2;
        i--;) {

        v[i] = qo[i] - po[i];
    }

    for (int i = 2;

```

```

        i--;) {

qpo[i] = 0.0;

for (int j = 2;
    j--;) {

    qpo[i] += pi[j][i] * v[j];
    qpc[j][i] = 0.0;

    for (int k = 2;
        k--;) {

        qpc[j][i] += pi[k][i] * qc[j][k];

    }
}
}
}

void Frame3DImportFrame(
    const Frame3D* const P,
    const Frame3D* const Q,
    Frame3D* const Qp) {

// Shortcuts
const double* qo = Q->orig;
double* qpo = Qp->orig;
const double* po = P->orig;

const double (*pi)[3] = P->invComp;
double (*qpc)[3] = Qp->comp;
const double (*qc)[3] = Q->comp;

// Calculate the projection
double v[3];
for (int i = 3;
    i--;) {

    v[i] = qo[i] - po[i];

}

for (int i = 3;
    i--;) {

    qpo[i] = 0.0;

    for (int j = 3;
        j--;) {

        qpo[i] += pi[j][i] * v[j];
        qpc[j][i] = 0.0;

        for (int k = 3;
            k--;) {

            qpc[j][i] += pi[k][i] * qc[j][k];

        }
    }
}
}
}

```

```

}

// Export the AABB 'bdgBox' from 'that' 's coordinates system to
// the real coordinates system and update 'bdgBox' with the resulting
// AABB
void Frame2DExportBdgBox(
    const Frame2D* const that,
    const AABB2D* const bdgBox,
    AABB2D* const bdgBoxProj) {

    // Shortcuts
    const double* to = that->orig;
    const double* bbmi = bdgBox->min;
    const double* bbma = bdgBox->max;
    double* bbpmi = bdgBoxProj->min;
    double* bbpma = bdgBoxProj->max;

    const double (*tc)[2] = that->comp;

    // Initialise the coordinates of the result AABB with the projection
    // of the first corner of the AABB in argument
    for (int i = 2;
        i--;) {

        bbpma[i] = to[i];

        for (int j = 2;
            j--;) {

            bbpma[i] += tc[j][i] * bbmi[j];

        }

        bbpmi[i] = bbpma[i];
    }

    // Loop on vertices of the AABB
    // skip the first vertex which is the origin already computed above
    int nbVertices = powi(2, 2);
    for (int iVertex = nbVertices;
        iVertex-- && iVertex;) {

        // Declare a variable to memorize the coordinates of the vertex in
        // 'that' 's coordinates system
        double v[2];

        // Calculate the coordinates of the vertex in
        // 'that' 's coordinates system
        for (int i = 2;
            i--;) {

            v[i] = ((iVertex & (1 << i)) ? bbma[i] : bbmi[i]);

        }

        // Declare a variable to memorize the projected coordinates
        // in real coordinates system
        double w[2];

        // Project the vertex to real coordinates system
        for (int i = 2;

```

```

        i--;) {

w[i] = to[i];

for (int j = 2;
    j--;) {

    w[i] += tc[j][i] * v[j];

}
}

// Update the coordinates of the result AABB
for (int i = 2;
    i--;) {

    if (bbpma[i] > w[i]) {

        bbpma[i] = w[i];

    }
    if (bbpma[i] < w[i]) {

        bbpma[i] = w[i];

    }
}
}

}

void Frame3DExportBdgBox(
    const Frame3D* const that,
    const AABB3D* const bdgBox,
    AABB3D* const bdgBoxProj) {

// Shortcuts
const double* to    = that->orig;
const double* bbmi  = bdgBox->min;
const double* bbma  = bdgBox->max;
double* bbpma      = bdgBoxProj->min;
double* bbpma      = bdgBoxProj->max;

const double (*tc)[3] = that->comp;

// Initialise the coordinates of the result AABB with the projection
// of the first corner of the AABB in argument
for (int i = 3;
    i--;) {

    bbpma[i] = to[i];

    for (int j = 3;
        j--;) {

        bbpma[i] += tc[j][i] * bbmi[j];

    }

    bbpma[i] = bbpma[i];

}
}

```

```

// Loop on vertices of the AABB
// skip the first vertex which is the origin already computed above
int nbVertices = powi(2, 3);
for (int iVertex = nbVertices;
     iVertex-- && iVertex;) {

    // Declare a variable to memorize the coordinates of the vertex in
    // 'that' 's coordinates system
    double v[3];

    // Calculate the coordinates of the vertex in
    // 'that' 's coordinates system
    for (int i = 3;
         i--;) {

        v[i] = ((iVertex & (1 << i)) ? bbma[i] : bbmi[i]);

    }

    // Declare a variable to memorize the projected coordinates
    // in real coordinates system
    double w[3];

    // Project the vertex to real coordinates system
    for (int i = 3;
         i--;) {

        w[i] = to[i];

        for (int j = 3;
             j--;) {

            w[i] += tc[j][i] * v[j];

        }

    }

    // Update the coordinates of the result AABB
    for (int i = 3;
         i--;) {

        if (bbpmi[i] > w[i]) {

            bbpmi[i] = w[i];

        }
        if (bbpma[i] < w[i]) {

            bbpma[i] = w[i];

        }

    }

}

}

// Print the AABB 'that' on stdout
// Output format is (min[0], min[1], ...)-(max[0], max[1], ...)
void AABB2DPrint(const AABB2D* const that) {

    printf("(");

```

```

    for (int i = 0;
        i < 2;
        ++i) {

        printf("%f", that->min[i]);
        if (i < 1)
            printf(",");

    }
    printf(")-(");
    for (int i = 0;
        i < 2;
        ++i) {

        printf("%f", that->max[i]);
        if (i < 1)
            printf(",");

    }
    printf(")");
}

void AABBB3DPrint(const AABBB3D* const that) {

    printf("(");
    for (int i = 0;
        i < 3;
        ++i) {

        printf("%f", that->min[i]);
        if (i < 2)
            printf(",");

    }
    printf(")-(");
    for (int i = 0;
        i < 3;
        ++i) {

        printf("%f", that->max[i]);
        if (i < 2)
            printf(",");

    }
    printf(")");
}

// Print the Frame 'that' on stdout
// Output format is (orig[0], orig[1], orig[2])
// (comp[0][0], comp[0][1], comp[0][2])
// (comp[1][0], comp[1][1], comp[1][2])
// (comp[2][0], comp[2][1], comp[2][2])
void Frame2DPrint(const Frame2D* const that) {
    if (that->type == FrameTetrahedron) {
        printf("T");
    } else if (that->type == FrameCuboid) {
        printf("C");
    }
    printf("(");
    for (int i = 0;

```

```

        i < 2;
        ++i) {

    printf("%f", that->orig[i]);
    if (i < 1)
        printf(",");

}
for (int j = 0;
     j < 2;
     ++j) {
    printf(") (");
    for (int i = 0;
         i < 2;
         ++i) {

        printf("%f", that->comp[j][i]);
        if (i < 1)
            printf(",");

    }
}
printf(")");
}

void Frame3DPrint(const Frame3D* const that) {
    if (that->type == FrameTetrahedron) {
        printf("T");
    } else if (that->type == FrameCuboid) {
        printf("C");
    }
    printf("(");
    for (int i = 0;
         i < 3;
         ++i) {

        printf("%f", that->orig[i]);
        if (i < 2)
            printf(",");

    }
    for (int j = 0;
         j < 3;
         ++j) {
        printf(") (");
        for (int i = 0;
             i < 3;
             ++i) {

            printf("%f", that->comp[j][i]);
            if (i < 2)
                printf(",");

        }
    }
    printf(")");
}

// Power function for integer base and exponent
// Return 'base' ^ 'exp'

```



```

int powi(
    int base,
    unsigned int exp) {

    int res = 1;
    for (;
        exp;
        --exp) {

        res *= base;

    }
    return res;
}

```

## 5.2 FMB

### 5.2.1 2D static

Header

```

#ifndef __FMB2D_H_
#define __FMB2D_H_

#include <stdbool.h>
#include "frame.h"

// ----- Functions declaration -----

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
bool FMBTestIntersection2D(
    const Frame2D* const that,
    const Frame2D* const tho,
    AABB2D* const bdgBox);

#endif

```

Body

```

#include "fmb2d.h"

// ----- Macros -----

// Return 1.0 if v is positive, -1.0 if v is negative, 0.0 else
#define sgn(v) (((0.0 < (v)) ? 1 : 0) - (((v) < 0.0) ? 1 : 0))

#define FST_VAR 0
#define SND_VAR 1
#define THD_VAR 2

```

```

#define EPSILON 0.0000001 //0.001

// ----- Functions declaration -----

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// Return false if the system becomes inconsistent during elimination,
// else return true
bool ElimVar2D(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[2],
    double* Yp,
    int* const nbRemainRows);

// Get the bounds of the 'iVar'-th variable in the 'nbRows' rows
// system 'M'.X<='Y' and store them in the 'iVar'-th axis of the
// AABB 'bdgBox'
// ('M' arrangement is [iRow][iCol])
// The system is supposed to have been reduced to only one variable
// per row, the one in argument, which can be located in a different
// column than 'iVar'
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBound2D(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    AABB2D* const bdgBox);

// ----- Functions implementation -----

void PrintMY2D(
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    const int nbVar) {
    for (int iRow = 0; iRow < nbRows; ++iRow) {
        for (int iCol = 0; iCol < nbVar; ++iCol) {
            printf("%f ", M[iRow][iCol]);
        }
        printf("| %f\n", Y[iRow]);
    }
}

void PrintM2D(
    const double (*M)[2],
    const int nbRows) {
    for (int iRow = 0; iRow < nbRows; ++iRow) {
        for (int iCol = 0; iCol < 2; ++iCol) {
            printf("%f ", M[iRow][iCol]);
        }
        printf("\n");
    }
}

```

```

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// ('M' arrangement is [iRow][iCol])
// Return true if the system becomes inconsistent during elimination,
// else return false
bool ElimVar2D(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[2],
    double* Yp,
    int* const nbRemainRows) {

    // Initialize the number of rows in the result system
    *nbRemainRows = 0;

    // First we process the rows where the eliminated variable is not null

    // For each row except the last one
    for (int iRow = 0;
        iRow < nbRows - 1;
        ++iRow) {

        // Shortcuts
        int sgnMIRowIVar = sgn(M[iRow][iVar]);
        double fabsMIRowIVar = fabs(M[iRow][iVar]);
        double YIRowDivideByFabsMIRowIVar = Y[iRow] / fabsMIRowIVar;

        // For each following rows
        for (int jRow = iRow + 1;
            jRow < nbRows;
            ++jRow) {

            // If coefficients of the eliminated variable in the two rows have
            // different signs and are not null
            if (sgnMIRowIVar != sgn(M[jRow][iVar]) &&
                fabsMIRowIVar > EPSILON &&
                fabs(M[jRow][iVar]) > EPSILON) {

                // Declare a variable to memorize the sum of the negative
                // coefficients in the row
                double sumNegCoeff = 0.0;

                // Declare a variable to memorize if all the coefficients
                // are >= 0.0
                bool allPositive = true;

                // Declare a variable to memorize if all the coefficients
                // are null
                bool allNull = true;

                // Add the sum of the two normed (relative to the eliminated
                // variable) rows into the result system. This actually
                // eliminate the variable while keeping the constraints on
                // others variables
                for (int iCol = 0, jCol = 0;
                    iCol < nbCols;
                    ++iCol) {

```

```

if (iCol != iVar) {

    Mp[*nbRemainRows][jCol] =
        M[iRow][iCol] / fabsMIRowIVar +
        M[jRow][iCol] / fabs(M[jRow][iVar]);

    // If the coefficient is negative
    if (Mp[*nbRemainRows][jCol] < -1.0 * EPSILON) {

        // Memorize that at least one coefficient is not positive
        allPositive = false;

        // Memorize that at least one coefficient is not null
        allNull = false;

        // Update the sum of the negative coefficient
        sumNegCoeff += Mp[*nbRemainRows][jCol];

        // Else, if the coefficient is positive
    } else if (Mp[*nbRemainRows][jCol] > EPSILON) {

        // Memorize that at least one coefficient is not null
        allNull = false;

    }

    ++jCol;

}

}

Yp[*nbRemainRows] =
    YIRowDivideByFabsMIRowIVar +
    Y[jRow] / fabs(M[jRow][iVar]);

// If at least one coefficient is not null
if (allNull == false) {

    // If all the coefficients are positive and the right side of
    // the inequality is negative
    if (allPositive == true &&
        Yp[*nbRemainRows] < 0.0) {

        // As X is in [0,1], the system is inconsistent
        return true;

    }

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Yp[*nbRemainRows] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        return true;

    }

    // Else all coefficients are null, if the right side is null
    // or negative
} else if (Yp[*nbRemainRows] <= 0.0) {

```

```

        // The system is inconsistent
        return true;

    }

    // Increment the nb of rows into the result system
    ++(*nbRemainRows);

}

}

}

// Then we copy and compress the rows where the eliminated
// variable is null

// Loop on rows of the input system
for (int iRow = 0;
     iRow < nbRows;
     ++iRow) {

    // Shortcut
    const double* MiRow = M[iRow];

    // If the coefficient of the eliminated variable is null on
    // this row
    if (fabs(M[iRow][iVar]) < EPSILON) {

        // Shortcut
        double* MpnbRemainRows = Mp[*nbRemainRows];

        // Copy this row into the result system excluding the eliminated
        // variable
        for (int iCol = 0, jCol = 0;
             iCol < nbCols;
             ++iCol) {

            if (iCol != iVar) {

                MpnbRemainRows[jCol] = MiRow[iCol];

                ++jCol;

            }

        }

        Yp[*nbRemainRows] = Y[iRow];

        // Increment the nb of rows into the result system
        ++(*nbRemainRows);

    }

}

return false;

}

```

```

// Get the bounds of the 'iVar'-th variable in the 'nbRows' rows
// system 'M'.X<='Y' and store them in the 'iVar'-th axis of the
// ABB 'bdgBox'
// ('M' arrangement is [iRow][iCol])
// The system is supposed to have been reduced to only one variable
// per row, the one in argument
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBound2D(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    ABB2D* const bdgBox) {

    // Shortcuts
    double* min = bdgBox->min + iVar;
    double* max = bdgBox->max + iVar;

    // Initialize the bounds to there maximum maximum and minimum minimum
    *min = 0.0;
    *max = 1.0;

    // Loop on rows
    for (int jRow = 0;
        jRow < nbRows;
        ++jRow) {

        // Shortcut
        double MjRowiVar = M[jRow][0];

        // If this row has been reduced to the variable in argument
        // and it has a strictly positive coefficient
        if (MjRowiVar > EPSILON) {

            // Get the scaled value of Y for this row
            double y = Y[jRow] / MjRowiVar;

            // If the value is lower than the current maximum bound
            if (*max > y) {

                // Update the maximum bound
                *max = y;

            }

            // Else, if this row has been reduced to the variable in argument
            // and it has a strictly negative coefficient
        } else if (MjRowiVar < -1.0 * EPSILON) {

            // Get the scaled value of Y for this row
            double y = Y[jRow] / MjRowiVar;

            // If the value is greater than the current minimum bound
            if (*min < y) {

                // Update the minimum bound
                *min = y;

            }

        }

    }

}

```

```

    }

}

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
bool FMBTestIntersection2D(
    const Frame2D* const that,
    const Frame2D* const tho,
    AABB2D* const bdgBox) {

    // Get the projection of the Frame 'tho' in Frame 'that' coordinates
    // system
    Frame2D thoProj;
    Frame2DImportFrame(that, tho, &thoProj);

    // Declare two variables to memorize the system to be solved M.X <= Y
    // (M arrangement is [iRow][iCol])
    double M[10][2];
    double Y[10];

    // Shortcuts
    double (*thoProjComp)[2] = thoProj.comp;
    double *thoProjOrig = thoProj.orig;

    // Variable to memorise the current row in the system
    int iRow = 0;

    // Shortcuts
    double* MIRow;

    // Constraints 0 + C.X >= 0.0
    // and constraints x_i >= 0.0
    for (;
        iRow < 2;
        ++iRow) {

        // Shortcuts
        MIRow = M[iRow];
        double* MJRow = M[iRow + 2];

        // For each column of the system
        double sumNeg = 0.0;
        for (int iCol = 2;
            iCol--;) {

            MIRow[iCol] = -1.0 * thoProjComp[iCol][iRow];
            if (MIRow[iCol] < 0.0) {
                sumNeg += MIRow[iCol];
            }

            // If it's on the diagonal
            if (iRow == iCol) {

```

```

        MJRow[iCol] = -1.0;

// Else it's not on the diagonal
} else {

        MJRow[iCol] = 0.0;

}
}
if (thoProjOrig[iRow] < sumNeg)
    return false;
Y[iRow] = thoProjOrig[iRow];
Y[iRow + 2] = 0.0;
}
iRow = 4;

if (that->type == FrameCuboid) {

// Constraints 0 + C.X <= 1.0
for (int jRow = 0;
    jRow < 2;
    ++jRow, ++iRow) {

// Shortcuts
MRow = M[iRow];
double* MJRow = M[jRow];

// For each column of the system
double sumNeg = 0.0;
for (int iCol = 2;
    iCol--;) {

    MRow[iCol] = -1.0 * MJRow[iCol];
    if (MRow[iCol] < 0.0) {
        sumNeg += MRow[iCol];
    }

}

Y[iRow] = 1.0 - thoProjOrig[jRow];
if (Y[iRow] < sumNeg)
    return false;
}

// Else, the first frame is a Tetrahedron
} else {

// Declare a variable to memorize the sum of the negative
// coefficients in the row
double sumNegCoeff = 0.0;

// Declare a variable to memorize if all the coefficients
// are null
bool allNull = true;

// Shortcut
double* MRow = M[iRow];

Y[iRow] = 1.0;

// For each column of the system
for (int iCol = 2;
    iCol--;) {

```



```

MRow[iCol] = 0.0;

// For each component
for (int iAxis = 2;
     iAxis--;) {

    MRow[iCol] += thoProjComp[iCol][iAxis];

}

Y[iRow] -= thoProjOrig[iCol];

// If the coefficient is negative
if (MRow[iCol] < -1.0 * EPSILON) {

    // Memorize that at least one coefficient is not null
    allNull = false;

    // Update the sum of the negative coefficient
    sumNegCoeff += MRow[iCol];

// Else, if the coefficient is positive
} else if (MRow[iCol] > EPSILON) {

    // Memorize that at least one coefficient is not null
    allNull = false;

}

}

// If at least one coefficient is not null
if (allNull == false) {

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[iRow] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        // there is no intersection
        return false;

    }

// Else all coefficients are null, if the right side is null
// or negative
} else if (Y[iRow] <= 0.0) {

    // The system is inconsistent, there is no intersection
    return false;

}

// Update the number of rows in the system
++iRow;

}

if (tho->type == FrameCuboid) {

    // Constraints  $x_i \leq 1.0$ 

```

```

for (int jRow = 0;
    jRow < 2;
    ++jRow, ++iRow) {

    // Shortcuts
    MRow = M[iRow];

    // For each column of the system
    for (int iCol = 2;
        iCol--;) {
        // If it's on the diagonal
        if (jRow == iCol) {

            MRow[iCol] = 1.0;

            // Else it's not on the diagonal
        } else {

            MRow[iCol] = 0.0;

        }
    }
    Y[iRow] = 1.0;
}

// Else, the second frame is a Tetrahedron
} else {

    // Declare a variable to memorize the sum of the negative
    // coefficients in the row
    double sumNegCoeff = 0.0;

    // Declare a variable to memorize if all the coefficients
    // are null
    bool allNull = true;

    // Shortcut
    double* MRow = M[iRow];

    // For each column of the system
    for (int iCol = 2;
        iCol--;) {

        MRow[iCol] = 1.0;

        // If the coefficient is negative
        if (MRow[iCol] < -1.0 * EPSILON) {

            // Memorize that at least one coefficient is not null
            allNull = false;

            // Update the sum of the negative coefficient
            sumNegCoeff += MRow[iCol];

            // Else, if the coefficient is positive
        } else if (MRow[iCol] > EPSILON) {

            // Memorize that at least one coefficient is not null
            allNull = false;

```

```

    }

}

Y[iRow] = 1.0;

// If at least one coefficient is not null
if (allNull == false) {

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[iRow] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        // there is no intersection
        return false;

    }

    // Else all coefficients are null, if the right side is null
    // or negative
    } else if (Y[iRow] <= 0.0) {

        // The system is inconsistent, there is no intersection
        return false;

    }

    // Update the number of rows in the system
    ++iRow;

}

// Declare a variable to memorize the total number of rows in the
// system. It may vary depending on the type of Frames
int nbRows = iRow;

// Solve the system

// Declare a AABB to memorize the bounding box of the intersection
// in the coordinates system of that
AABB2D bdgBoxLocal;

// Declare variables to eliminate the first variable
// The number of rows is set conservatively, one may try to reduce
// them if needed
double Mp[60][2];
double Yp[60];
int nbRowsP;

// Eliminate the first variable
bool inconsistency =
    ElimVar2D(
        FST_VAR,
        M,
        Y,
        nbRows,
        2,
        Mp,
        Yp,
        &nbRowsP);

```

```

// If the system is inconsistent
if (inconsistency == true) {

    // The two Frames are not in intersection
    return false;

}

// Get the bounds for the remaining second variable
GetBound2D(
    SND_VAR,
    Mp,
    Yp,
    nbRowsP,
    &bdgBoxLocal);

// If the bounds are inconstant
if (bdgBoxLocal.min[SND_VAR] >= bdgBoxLocal.max[SND_VAR]) {

    // The two Frames are not in intersection
    return false;

// Else, if the bounds are consistent here it means
// the two Frames are in intersection.
// If the user hasn't requested for the resulting bounding box
} else if (bdgBox == NULL) {

    // Immediately return true
    return true;

}

// Now starts again from the initial systems and eliminate the
// second variable to get the bounds of the first variable
inconsistency =
    ElimVar2D(
        SND_VAR,
        M,
        Y,
        nbRows,
        2,
        Mp,
        Yp,
        &nbRowsP);
if (inconsistency == true) {
    return false;
}

GetBound2D(
    FST_VAR,
    Mp,
    Yp,
    nbRowsP,
    &bdgBoxLocal);

if (bdgBoxLocal.min[FST_VAR] >= bdgBoxLocal.max[FST_VAR]) {
    return false;
}

// If the user requested the resulting bounding box
if (bdgBox != NULL) {

```

```

// Export the local bounding box toward the real coordinates
// system
Frame2DExportBdgBox(
    tho,
    &bdgBoxLocal,
    bdgBox);

// Clip with the AABB of 'that'
double* const min = bdgBox->min;
double* const max = bdgBox->max;
const double* const thatBdgBoxMin = that->bdgBox.min;
const double* const thatBdgBoxMax = that->bdgBox.max;
for (int iAxis = 2;
     iAxis--;) {

    if (min[iAxis] < thatBdgBoxMin[iAxis]) {

        min[iAxis] = thatBdgBoxMin[iAxis];

    }
    if (max[iAxis] > thatBdgBoxMax[iAxis]) {

        max[iAxis] = thatBdgBoxMax[iAxis];

    }

}

}

// If we've reached here the two Frames are intersecting
return true;

}

```

## 5.2.2 3D static

### Header

```

#ifndef __FMB3D_H_
#define __FMB3D_H_

#include <stdbool.h>
#include "frame.h"

// ----- Functions declaration -----

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
bool FMBTestIntersection3D(
    const Frame3D* const that,
    const Frame3D* const tho,

```

```

        AAB3D* const bdgBox);

#endif

Body

#include "fmb3d.h"

// ----- Macros -----

// Return 1.0 if v is positive, -1.0 if v is negative, 0.0 else
#define sgn(v) (((0.0 < (v)) ? 1 : 0) - (((v) < 0.0) ? 1 : 0))

#define FST_VAR 0
#define SND_VAR 1
#define THD_VAR 2

#define EPSILON 0.0000001 //0.001

// ----- Functions declaration -----

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// Return false if the system becomes inconsistent during elimination,
// else return true
bool ElimVar3D(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[3],
    double* Yp,
    int* const nbRemainRows);

// Get the bounds of the 'iVar'-th variable in the 'nbRows' rows
// system 'M'.X<='Y' and store them in the 'iVar'-th axis of the
// AAB3D 'bdgBox'
// ('M' arrangement is [iRow][iCol])
// The system is supposed to have been reduced to only one variable
// per row, the one in argument, which can be located in a different
// column than 'iVar'
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBound3D(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    AAB3D* const bdgBox);

// ----- Functions implementation -----

void PrintMY3D(
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbVar) {
    for (int iRow = 0; iRow < nbRows; ++iRow) {

```

```

        for (int iCol = 0; iCol < nbVar; ++iCol) {
            printf("%f ", M[iRow][iCol]);
        }
        printf("| %f\n", Y[iRow]);
    }
}

void PrintM3D(
    const double (*M)[3],
    const int nbRows) {
    for (int iRow = 0; iRow < nbRows; ++iRow) {
        for (int iCol = 0; iCol < 3; ++iCol) {
            printf("%f ", M[iRow][iCol]);
        }
        printf("\n");
    }
}

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// ('M' arrangement is [iRow][iCol])
// Return true if the system becomes inconsistent during elimination,
// else return false
bool ElimVar3D(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[3],
    double* Yp,
    int* const nbRemainRows) {

    // Initialize the number of rows in the result system
    *nbRemainRows = 0;

    // First we process the rows where the eliminated variable is not null

    // For each row except the last one
    for (int iRow = 0;
        iRow < nbRows - 1;
        ++iRow) {

        // Shortcuts
        int sgnMIRowIVar = sgn(M[iRow][iVar]);
        double fabsMIRowIVar = fabs(M[iRow][iVar]);
        double YIRowDivideByFabsMIRowIVar = Y[iRow] / fabsMIRowIVar;

        // For each following rows
        for (int jRow = iRow + 1;
            jRow < nbRows;
            ++jRow) {

            // If coefficients of the eliminated variable in the two rows have
            // different signs and are not null
            if (sgnMIRowIVar != sgn(M[jRow][iVar]) &&
                fabsMIRowIVar > EPSILON &&
                fabs(M[jRow][iVar]) > EPSILON) {

                // Declare a variable to memorize the sum of the negative

```

```

// coefficients in the row
double sumNegCoeff = 0.0;

// Declare a variable to memorize if all the coefficients
// are >= 0.0
bool allPositive = true;

// Declare a variable to memorize if all the coefficients
// are null
bool allNull = true;

// Add the sum of the two normed (relative to the eliminated
// variable) rows into the result system. This actually
// eliminate the variable while keeping the constraints on
// others variables
for (int iCol = 0, jCol = 0;
    iCol < nbCols;
    ++iCol ) {

    if (iCol != iVar) {

        Mp[*nbRemainRows][jCol] =
            M[iRow][iCol] / fabsMIRowIVar +
            M[jRow][iCol] / fabs(M[jRow][iVar]);

        // If the coefficient is negative
        if (Mp[*nbRemainRows][jCol] < -1.0 * EPSILON) {

            // Memorize that at least one coefficient is not positive
            allPositive = false;

            // Memorize that at least one coefficient is not null
            allNull = false;

            // Update the sum of the negative coefficient
            sumNegCoeff += Mp[*nbRemainRows][jCol];

            // Else, if the coefficient is positive
        } else if (Mp[*nbRemainRows][jCol] > EPSILON) {

            // Memorize that at least one coefficient is not null
            allNull = false;

        }

        ++jCol;
    }
}

Yp[*nbRemainRows] =
    YIRowDivideByFabsMIRowIVar +
    Y[jRow] / fabs(M[jRow][iVar]);

// If at least one coefficient is not null
if (allNull == false) {

    // If all the coefficients are positive and the right side of
    // the inequality is negative
    if (allPositive == true &&
        Yp[*nbRemainRows] < 0.0) {

```



```

        // As X is in [0,1], the system is inconsistent
        return true;

    }

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Yp[*nbRemainRows] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        return true;

    }

    // Else all coefficients are null, if the right side is null
    // or negative
    } else if (Yp[*nbRemainRows] <= 0.0) {

        // The system is inconsistent
        return true;

    }

    // Increment the nb of rows into the result system
    ++(*nbRemainRows);

}

}

}

// Then we copy and compress the rows where the eliminated
// variable is null

// Loop on rows of the input system
for (int iRow = 0;
     iRow < nbRows;
     ++iRow) {

    // Shortcut
    const double* MiRow = M[iRow];

    // If the coefficient of the eliminated variable is null on
    // this row
    if (fabs(M[iRow][iVar]) < EPSILON) {

        // Shortcut
        double* MpnbRemainRows = Mp[*nbRemainRows];

        // Copy this row into the result system excluding the eliminated
        // variable
        for (int iCol = 0, jCol = 0;
             iCol < nbCols;
             ++iCol) {

            if (iCol != iVar) {

                MpnbRemainRows[jCol] = MiRow[iCol];

                ++jCol;
            }
        }
    }
}

```

```

    }

}

Yp[*nbRemainRows] = Y[iRow];

// Increment the nb of rows into the result system
++(*nbRemainRows);

}

}

return false;

}

// Get the bounds of the 'iVar'-th variable in the 'nbRows' rows
// system 'M'.X<='Y' and store them in the 'iVar'-th axis of the
// AABB 'bdgBox'
// ('M' arrangement is [iRow][iCol])
// The system is supposed to have been reduced to only one variable
// per row, the one in argument
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBound3D(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    AABB3D* const bdgBox) {

    // Shortcuts
    double* min = bdgBox->min + iVar;
    double* max = bdgBox->max + iVar;

    // Initialize the bounds to there maximum maximum and minimum minimum
    *min = 0.0;
    *max = 1.0;

    // Loop on rows
    for (int jRow = 0;
        jRow < nbRows;
        ++jRow) {

        // Shortcut
        double MjRowiVar = M[jRow][0];

        // If this row has been reduced to the variable in argument
        // and it has a strictly positive coefficient
        if (MjRowiVar > EPSILON) {

            // Get the scaled value of Y for this row
            double y = Y[jRow] / MjRowiVar;

            // If the value is lower than the current maximum bound
            if (*max > y) {

                // Update the maximum bound
                *max = y;
            }
        }
    }
}

```

```

    }

    // Else, if this row has been reduced to the variable in argument
    // and it has a strictly negative coefficient
    } else if (MjRowiVar < -1.0 * EPSILON) {

        // Get the scaled value of Y for this row
        double y = Y[jRow] / MjRowiVar;

        // If the value is greater than the current minimum bound
        if (*min < y) {

            // Update the minimum bound
            *min = y;

        }

    }

}

}

}

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
bool FMBTestIntersection3D(
    const Frame3D* const that,
    const Frame3D* const tho,
    AABB3D* const bdgBox) {

    // Get the projection of the Frame 'tho' in Frame 'that' coordinates
    // system
    Frame3D thoProj;
    Frame3DImportFrame(that, tho, &thoProj);

    // Declare two variables to memorize the system to be solved  $M.X \leq Y$ 
    // (M arrangement is [iRow][iCol])
    double M[14][3];
    double Y[14];

    // Shortcuts
    double (*thoProjComp)[3] = thoProj.comp;
    double *thoProjOrig = thoProj.orig;

    // Variable to memorise the current row in the system
    int iRow = 0;

    // Shortcuts
    double* MIRow;

    // Constraints  $0 + C.X \geq 0.0$ 
    // and constraints  $x_i \geq 0.0$ 
    for (;
        iRow < 3;
        ++iRow) {

```

```

// Shortcuts
MIRow = M[iRow];
double* MJRow = M[iRow + 3];

// For each column of the system
double sumNeg = 0.0;
for (int iCol = 3;
     iCol--;) {

    MIRow[iCol] = -1.0 * thoProjComp[iCol][iRow];
    if (MIRow[iCol] < 0.0) {
        sumNeg += MIRow[iCol];
    }

    // If it's on the diagonal
    if (iRow == iCol) {

        MJRow[iCol] = -1.0;

    // Else it's not on the diagonal
    } else {

        MJRow[iCol] = 0.0;

    }
}
if (thoProjOrig[iRow] < sumNeg)
    return false;
Y[iRow] = thoProjOrig[iRow];
Y[iRow + 3] = 0.0;
}
iRow = 6;

if (that->type == FrameCuboid) {

    // Constraints 0 + C.X <= 1.0
    for (int jRow = 0;
         jRow < 3;
         ++jRow, ++iRow) {

        // Shortcuts
        MIRow = M[iRow];
        double* MJRow = M[jRow];

        // For each column of the system
        double sumNeg = 0.0;
        for (int iCol = 3;
             iCol--;) {

            MIRow[iCol] = -1.0 * MJRow[iCol];
            if (MIRow[iCol] < 0.0) {
                sumNeg += MIRow[iCol];
            }

        }

        Y[iRow] = 1.0 - thoProjOrig[jRow];
        if (Y[iRow] < sumNeg)
            return false;
    }
}

// Else, the first frame is a Tetrahedron

```

```

} else {

    // Declare a variable to memorize the sum of the negative
    // coefficients in the row
    double sumNegCoeff = 0.0;

    // Declare a variable to memorize if all the coefficients
    // are null
    bool allNull = true;

    // Shortcut
    double* MRow = M[iRow];

    Y[iRow] = 1.0;

    // For each column of the system
    for (int iCol = 3;
        iCol--;) {

        MRow[iCol] = 0.0;

        // For each component
        for (int iAxis = 3;
            iAxis--;) {

            MRow[iCol] += thoProjComp[iCol][iAxis];

        }

        Y[iRow] -= thoProjOrig[iCol];

        // If the coefficient is negative
        if (MRow[iCol] < -1.0 * EPSILON) {

            // Memorize that at least one coefficient is not null
            allNull = false;

            // Update the sum of the negative coefficient
            sumNegCoeff += MRow[iCol];

        } else if (MRow[iCol] > EPSILON) {

            // Memorize that at least one coefficient is not null
            allNull = false;

        }

    }

    // If at least one coefficient is not null
    if (allNull == false) {

        // If the right side of the inequality is lower than the sum of
        // negative coefficients in the row
        if (Y[iRow] < sumNegCoeff) {

            // As X is in [0,1], the system is inconsistent
            // there is no intersection
            return false;

        }

    }

}

```

```

// Else all coefficients are null, if the right side is null
// or negative
} else if (Y[iRow] <= 0.0) {

    // The system is inconsistent, there is no intersection
    return false;

}

// Update the number of rows in the system
++iRow;

}

if (tho->type == FrameCuboid) {

    // Constraints  $x_i \leq 1.0$ 
    for (int jRow = 0;
        jRow < 3;
        ++jRow, ++iRow) {

        // Shortcuts
        MRow = M[iRow];

        // For each column of the system
        for (int iCol = 3;
            iCol--;) {
            // If it's on the diagonal
            if (jRow == iCol) {

                MRow[iCol] = 1.0;

            // Else it's not on the diagonal
            } else {

                MRow[iCol] = 0.0;

            }

        }

        Y[iRow] = 1.0;

    }

// Else, the second frame is a Tetrahedron
} else {

    // Declare a variable to memorize the sum of the negative
    // coefficients in the row
    double sumNegCoeff = 0.0;

    // Declare a variable to memorize if all the coefficients
    // are null
    bool allNull = true;

    // Shortcut
    double* MRow = M[iRow];

    // For each column of the system
    for (int iCol = 3;
        iCol--;) {

```

```

MRow[iCol] = 1.0;

// If the coefficient is negative
if (MRow[iCol] < -1.0 * EPSILON) {

    // Memorize that at least one coefficient is not null
    allNull = false;

    // Update the sum of the negative coefficient
    sumNegCoeff += MRow[iCol];

// Else, if the coefficient is positive
} else if (MRow[iCol] > EPSILON) {

    // Memorize that at least one coefficient is not null
    allNull = false;

}

}

Y[iRow] = 1.0;

// If at least one coefficient is not null
if (allNull == false) {

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[iRow] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        // there is no intersection
        return false;

    }

// Else all coefficients are null, if the right side is null
// or negative
} else if (Y[iRow] <= 0.0) {

    // The system is inconsistent, there is no intersection
    return false;

}

// Update the number of rows in the system
++iRow;

}

// Declare a variable to memorize the total number of rows in the
// system. It may vary depending on the type of Frames
int nbRows = iRow;

// Solve the system

// Declare a AABB to memorize the bounding box of the intersection
// in the coordinates system of that
AABB3D bdgBoxLocal;

// Declare variables to eliminate the first variable

```

```

// The number of rows is set conservatively, one may try to reduce
// them if needed
double Mp[64][3];
double Yp[64];
int nbRowsP;

// Eliminate the first variable
bool inconsistency =
    ElimVar3D(
        FST_VAR,
        M,
        Y,
        nbRows,
        3,
        Mp,
        Yp,
        &nbRowsP);

// If the system is inconsistent
if (inconsistency == true) {

    // The two Frames are not in intersection
    return false;

}

// Declare variables to eliminate the second variable
// The number of rows is set conservatively, one may try to reduce
// them if needed
double Mpp[514][3];
double Ypp[514];
int nbRowsPP;

// Eliminate the second variable (which is the first in the new system)
inconsistency =
    ElimVar3D(
        FST_VAR,
        Mp,
        Yp,
        nbRowsP,
        2,
        Mpp,
        Ypp,
        &nbRowsPP);

// If the system is inconsistent
if (inconsistency == true) {

    // The two Frames are not in intersection
    return false;

}

// Get the bounds for the remaining third variable
GetBound3D(
    THD_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

// If the bounds are inconstant

```



```

if (bdgBoxLocal.min[THD_VAR] >= bdgBoxLocal.max[THD_VAR]) {

    // The two Frames are not in intersection
    return false;

}

// Eliminate the third variable (which is second first in the new
// system)
inconsistency =
    ElimVar3D(
        SND_VAR,
        Mp,
        Yp,
        nbRowsP,
        2,
        Mpp,
        Ypp,
        &nbRowsPP);

// If the resulting system is inconsistent
if (inconsistency == true) {

    // The two Frames are not in intersection
    return false;

// Else, if the bounds are consistent here it means
// the two Frames are in intersection.
// If the user hasn't requested for the resulting bounding box
} else if (bdgBox == NULL) {

    // Immediately return true
    return true;

}

// Get the bounds for the remaining second variable
GetBound3D(
    SND_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

// If the bounds are inconstant
if (bdgBoxLocal.min[SND_VAR] >= bdgBoxLocal.max[SND_VAR]) {

    // The two Frames are not in intersection
    return false;

}

// Now starts again from the initial systems and eliminate the
// second and third variables to get the bounds of the first variable
inconsistency =
    ElimVar3D(
        SND_VAR,
        M,
        Y,
        nbRows,
        3,
        Mp,

```

```

        Yp,
        &nbRowsP);

if (inconsistency == true) {
    return false;
}

inconsistency =
    ElimVar3D(
        SND_VAR,
        Mp,
        Yp,
        nbRowsP,
        2,
        Mpp,
        Ypp,
        &nbRowsPP);

if (inconsistency == true) {
    return false;
}

GetBound3D(
    FST_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

if (bdgBoxLocal.min[FST_VAR] >= bdgBoxLocal.max[FST_VAR]) {
    return false;
}

// If the user requested the resulting bounding box
if (bdgBox != NULL) {

    // Export the local bounding box toward the real coordinates
    // system
    Frame3DExportBdgBox(
        tho,
        &bdgBoxLocal,
        bdgBox);

    // Clip with the AABB of 'that'
    double* const min = bdgBox->min;
    double* const max = bdgBox->max;
    const double* const thatBdgBoxMin = that->bdgBox.min;
    const double* const thatBdgBoxMax = that->bdgBox.max;
    for (int iAxis = 3;
        iAxis--;) {

        if (min[iAxis] < thatBdgBoxMin[iAxis]) {

            min[iAxis] = thatBdgBoxMin[iAxis];

        }
        if (max[iAxis] > thatBdgBoxMax[iAxis]) {

            max[iAxis] = thatBdgBoxMax[iAxis];

        }
    }
}

```

```

    }

}

// If we've reached here the two Frames are intersecting
return true;

}

```

### 5.2.3 2D dynamic

#### Header

```

#ifndef __FMB2D_H_
#define __FMB2D_H_

#include <stdbool.h>
#include "frame.h"

// ----- Functions declaration -----

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
bool FMBTestIntersection2DTime(
    const Frame2D* const that,
    const Frame2D* const tho,
    AABB2D* const bdgBox);

#endif

```

#### Body

```

#include "fmb2dt.h"

// ----- Macros -----

// Return 1.0 if v is positive, -1.0 if v is negative, 0.0 else
#define sgn(v) (((0.0 < (v)) ? 1 : 0) - (((v) < 0.0) ? 1 : 0))

#define FST_VAR 0
#define SND_VAR 1
#define THD_VAR 2

#define EPSILON 0.0000001 //0.001

// ----- Functions declaration -----

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// Return false if the system becomes inconsistent during elimination,

```

```

// else return true
bool ElimVar2DTime(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[2],
    double* Yp,
    int* const nbRemainRows);

// Get the bounds of the 'iVar'-th variable in the 'nbRows' rows
// system 'M'.X<='Y' and store them in the 'iVar'-th axis of the
// AABB 'bdgBox'
// ('M' arrangement is [iRow][iCol])
// The system is supposed to have been reduced to only one variable
// per row, the one in argument, which can be located in a different
// column than 'iVar'
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBound2DTime(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    AABB2D* const bdgBox);

// ----- Functions implementation -----

void PrintMY2DTime(
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    const int nbVar) {
    for (int iRow = 0; iRow < nbRows; ++iRow) {
        for (int iCol = 0; iCol < nbVar; ++iCol) {
            printf("%f ", M[iRow][iCol]);
        }
        printf("| %f\n", Y[iRow]);
    }
}

void PrintM2DTime(
    const double (*M)[2],
    const int nbRows) {
    for (int iRow = 0; iRow < nbRows; ++iRow) {
        for (int iCol = 0; iCol < 2; ++iCol) {
            printf("%f ", M[iRow][iCol]);
        }
        printf("\n");
    }
}

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// ('M' arrangement is [iRow][iCol])
// Return true if the system becomes inconsistent during elimination,
// else return false
bool ElimVar2DTime(
    const int iVar,

```

```

const double (*M)[2],
const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[2],
    double* Yp,
    int* const nbRemainRows) {

// Initialize the number of rows in the result system
*nbRemainRows = 0;

// First we process the rows where the eliminated variable is not null

// For each row except the last one
for (int iRow = 0;
    iRow < nbRows - 1;
    ++iRow) {

    // Shortcuts
    int sgnMIRowIVar = sgn(M[iRow][iVar]);
    double fabsMIRowIVar = fabs(M[iRow][iVar]);
    double YIRowDivideByFabsMIRowIVar = Y[iRow] / fabsMIRowIVar;

    // For each following rows
    for (int jRow = iRow + 1;
        jRow < nbRows;
        ++jRow) {

        // If coefficients of the eliminated variable in the two rows have
        // different signs and are not null
        if (sgnMIRowIVar != sgn(M[jRow][iVar]) &&
            fabsMIRowIVar > EPSILON &&
            fabs(M[jRow][iVar]) > EPSILON) {

            // Declare a variable to memorize the sum of the negative
            // coefficients in the row
            double sumNegCoeff = 0.0;

            // Declare a variable to memorize if all the coefficients
            // are >= 0.0
            bool allPositive = true;

            // Declare a variable to memorize if all the coefficients
            // are null
            bool allNull = true;

            // Add the sum of the two normed (relative to the eliminated
            // variable) rows into the result system. This actually
            // eliminate the variable while keeping the constraints on
            // others variables
            for (int iCol = 0, jCol = 0;
                iCol < nbCols;
                ++iCol) {

                if (iCol != iVar) {

                    Mp[*nbRemainRows][jCol] =
                        M[iRow][iCol] / fabsMIRowIVar +
                        M[jRow][iCol] / fabs(M[jRow][iVar]);

                    // If the coefficient is negative
                    if (Mp[*nbRemainRows][jCol] < -1.0 * EPSILON) {

```

```

        // Memorize that at least one coefficient is not positive
        allPositive = false;

        // Memorize that at least one coefficient is not null
        allNull = false;

        // Update the sum of the negative coefficient
        sumNegCoeff += Mp[*nbRemainRows][jCol];

        // Else, if the coefficient is positive
    } else if (Mp[*nbRemainRows][jCol] > EPSILON) {

        // Memorize that at least one coefficient is not null
        allNull = false;

    }

    ++jCol;

}

}

Yp[*nbRemainRows] =
    YIRowDivideByFabsMIRowIVar +
    Y[jRow] / fabs(M[jRow][iVar]);

// If at least one coefficient is not null
if (allNull == false) {

    // If all the coefficients are positive and the right side of
    // the inequality is negative
    if (allPositive == true &&
        Yp[*nbRemainRows] < 0.0) {

        // As X is in [0,1], the system is inconsistent
        return true;

    }

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Yp[*nbRemainRows] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        return true;

    }

    // Else all coefficients are null, if the right side is null
    // or negative
    } else if (Yp[*nbRemainRows] <= 0.0) {

        // The system is inconsistent
        return true;

    }

    // Increment the nb of rows into the result system
    ++(*nbRemainRows);

```

```

    }

}

}

// Then we copy and compress the rows where the eliminated
// variable is null

// Loop on rows of the input system
for (int iRow = 0;
     iRow < nbRows;
     ++iRow) {

    // Shortcut
    const double* MiRow = M[iRow];

    // If the coefficient of the eliminated variable is null on
    // this row
    if (fabs(M[iRow][iVar]) < EPSILON) {

        // Shortcut
        double* MpnbRemainRows = Mp[*nbRemainRows];

        // Copy this row into the result system excluding the eliminated
        // variable
        for (int iCol = 0, jCol = 0;
             iCol < nbCols;
             ++iCol) {

            if (iCol != iVar) {

                MpnbRemainRows[jCol] = MiRow[iCol];

                ++jCol;
            }
        }

        Yp[*nbRemainRows] = Y[iRow];

        // Increment the nb of rows into the result system
        ++(*nbRemainRows);
    }
}

return false;
}

// Get the bounds of the 'iVar'-th variable in the 'nbRows' rows
// system 'M'.X<='Y' and store them in the 'iVar'-th axis of the
// AABB 'bdgBox'
// ('M' arrangement is [iRow][iCol])
// The system is supposed to have been reduced to only one variable
// per row, the one in argument
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBound2DTime(

```

```

    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    AABB2D* const bdgBox) {

// Shortcuts
double* min = bdgBox->min + iVar;
double* max = bdgBox->max + iVar;

// Initialize the bounds to there maximum maximum and minimum minimum
*min = 0.0;
*max = 1.0;

// Loop on rows
for (int jRow = 0;
    jRow < nbRows;
    ++jRow) {

    // Shortcut
    double MjRowiVar = M[jRow][0];

    // If this row has been reduced to the variable in argument
    // and it has a strictly positive coefficient
    if (MjRowiVar > EPSILON) {

        // Get the scaled value of Y for this row
        double y = Y[jRow] / MjRowiVar;

        // If the value is lower than the current maximum bound
        if (*max > y) {

            // Update the maximum bound
            *max = y;

        }

        // Else, if this row has been reduced to the variable in argument
        // and it has a strictly negative coefficient
    } else if (MjRowiVar < -1.0 * EPSILON) {

        // Get the scaled value of Y for this row
        double y = Y[jRow] / MjRowiVar;

        // If the value is greater than the current minimum bound
        if (*min < y) {

            // Update the minimum bound
            *min = y;

        }

    }

}

}

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified

```



```

// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
bool FMBTestIntersection2DTime(
    const Frame2D* const that,
    const Frame2D* const tho,
    AABB2D* const bdgBox) {

    // Get the projection of the Frame 'tho' in Frame 'that' coordinates
    // system
    Frame2D thoProj;
    Frame2DImportFrame(that, tho, &thoProj);

    // Declare two variables to memorize the system to be solved M.X <= Y
    // (M arrangement is [iRow][iCol])
    double M[10][2];
    double Y[10];

    // Shortcuts
    double (*thoProjComp)[2] = thoProj.comp;
    double *thoProjOrig = thoProj.orig;

    // Variable to memorise the current row in the system
    int iRow = 0;

    // Shortcuts
    double* MIRow;

    // Constraints 0 + C.X >= 0.0
    // and constraints x_i >= 0.0
    for (;
        iRow < 2;
        ++iRow) {

        // Shortcuts
        MIRow = M[iRow];
        double* MJRow = M[iRow + 2];

        // For each column of the system
        double sumNeg = 0.0;
        for (int iCol = 2;
            iCol--;) {

            MIRow[iCol] = -1.0 * thoProjComp[iCol][iRow];
            if (MIRow[iCol] < 0.0) {
                sumNeg += MIRow[iCol];
            }

            // If it's on the diagonal
            if (iRow == iCol) {

                MJRow[iCol] = -1.0;

            // Else it's not on the diagonal
            } else {

                MJRow[iCol] = 0.0;
            }
        }
    }
}

```

```

        if (thoProjOrig[iRow] < sumNeg)
            return false;
        Y[iRow] = thoProjOrig[iRow];
        Y[iRow + 2] = 0.0;
    }
    iRow = 4;

    if (that->type == FrameCuboid) {

        // Constraints 0 + C.X <= 1.0
        for (int jRow = 0;
            jRow < 2;
            ++jRow, ++iRow) {

            // Shortcuts
            MIRow = M[iRow];
            double* MJRow = M[jRow];

            // For each column of the system
            double sumNeg = 0.0;
            for (int iCol = 2;
                iCol--;) {

                MIRow[iCol] = -1.0 * MJRow[iCol];
                if (MIRow[iCol] < 0.0) {
                    sumNeg += MIRow[iCol];
                }

            }
            Y[iRow] = 1.0 - thoProjOrig[jRow];
            if (Y[iRow] < sumNeg)
                return false;
        }

        // Else, the first frame is a Tetrahedron
    } else {

        // Declare a variable to memorize the sum of the negative
        // coefficients in the row
        double sumNegCoeff = 0.0;

        // Declare a variable to memorize if all the coefficients
        // are null
        bool allNull = true;

        // Shortcut
        double* MRow = M[iRow];

        Y[iRow] = 1.0;

        // For each column of the system
        for (int iCol = 2;
            iCol--;) {

            MRow[iCol] = 0.0;

            // For each component
            for (int iAxis = 2;
                iAxis--;) {

                MRow[iCol] += thoProjComp[iCol][iAxis];
            }
        }
    }
}

```

```

    }

    Y[iRow] -= thoProjOrig[iCol];

    // If the coefficient is negative
    if (MRow[iCol] < -1.0 * EPSILON) {

        // Memorize that at least one coefficient is not null
        allNull = false;

        // Update the sum of the negative coefficient
        sumNegCoeff += MRow[iCol];

    // Else, if the coefficient is positive
    } else if (MRow[iCol] > EPSILON) {

        // Memorize that at least one coefficient is not null
        allNull = false;

    }

}

// If at least one coefficient is not null
if (allNull == false) {

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[iRow] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        // there is no intersection
        return false;

    }

    // Else all coefficients are null, if the right side is null
    // or negative
    } else if (Y[iRow] <= 0.0) {

        // The system is inconsistent, there is no intersection
        return false;

    }

    // Update the number of rows in the system
    ++iRow;

}

if (tho->type == FrameCuboid) {

    // Constraints  $x_i \leq 1.0$ 
    for (int jRow = 0;
        jRow < 2;
        ++jRow, ++iRow) {

        // Shortcuts
        MRow = M[iRow];

        // For each column of the system
        for (int iCol = 2;

```

```

        iCol--;) {
// If it's on the diagonal
if (jRow == iCol) {

    MRow[iCol] = 1.0;

// Else it's not on the diagonal
} else {

    MRow[iCol] = 0.0;

}

}

Y[iRow] = 1.0;

}

// Else, the second frame is a Tetrahedron
} else {

// Declare a variable to memorize the sum of the negative
// coefficients in the row
double sumNegCoeff = 0.0;

// Declare a variable to memorize if all the coefficients
// are null
bool allNull = true;

// Shortcut
double* MRow = M[iRow];

// For each column of the system
for (int iCol = 2;
    iCol--;) {

    MRow[iCol] = 1.0;

// If the coefficient is negative
if (MRow[iCol] < -1.0 * EPSILON) {

// Memorize that at least one coefficient is not null
allNull = false;

// Update the sum of the negative coefficient
sumNegCoeff += MRow[iCol];

// Else, if the coefficient is positive
} else if (MRow[iCol] > EPSILON) {

// Memorize that at least one coefficient is not null
allNull = false;

}

}

Y[iRow] = 1.0;

// If at least one coefficient is not null
if (allNull == false) {

```

```

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[iRow] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        // there is no intersection
        return false;

    }

    // Else all coefficients are null, if the right side is null
    // or negative
    } else if (Y[iRow] <= 0.0) {

        // The system is inconsistent, there is no intersection
        return false;

    }

    // Update the number of rows in the system
    ++iRow;

}

// Declare a variable to memorize the total number of rows in the
// system. It may vary depending on the type of Frames
int nbRows = iRow;

// Solve the system

// Declare a AABB to memorize the bounding box of the intersection
// in the coordinates system of that
AABB2D bdgBoxLocal;

// Declare variables to eliminate the first variable
// The number of rows is set conservatively, one may try to reduce
// them if needed
double Mp[60][2];
double Yp[60];
int nbRowsP;

// Eliminate the first variable
bool inconsistency =
    ElimVar2DTime(
        FST_VAR,
        M,
        Y,
        nbRows,
        2,
        Mp,
        Yp,
        &nbRowsP);

// If the system is inconsistent
if (inconsistency == true) {

    // The two Frames are not in intersection
    return false;

}

// Get the bounds for the remaining second variable

```

```

GetBound2DTime(
    SND_VAR,
    Mp,
    Yp,
    nbRowsP,
    &bdgBoxLocal);

// If the bounds are inconstant
if (bdgBoxLocal.min[SND_VAR] >= bdgBoxLocal.max[SND_VAR]) {

    // The two Frames are not in intersection
    return false;

// Else, if the bounds are consistent here it means
// the two Frames are in intersection.
// If the user hasn't requested for the resulting bounding box
} else if (bdgBox == NULL) {

    // Immediately return true
    return true;

}

// Now starts again from the initial systems and eliminate the
// second variable to get the bounds of the first variable
inconsistency =
    ElimVar2DTime(
        SND_VAR,
        M,
        Y,
        nbRows,
        2,
        Mp,
        Yp,
        &nbRowsP);
if (inconsistency == true) {
    return false;
}

GetBound2DTime(
    FST_VAR,
    Mp,
    Yp,
    nbRowsP,
    &bdgBoxLocal);

if (bdgBoxLocal.min[FST_VAR] >= bdgBoxLocal.max[FST_VAR]) {
    return false;
}

// If the user requested the resulting bounding box
if (bdgBox != NULL) {

    // Export the local bounding box toward the real coordinates
    // system
    Frame2DExportBdgBox(
        tho,
        &bdgBoxLocal,
        bdgBox);

    // Clip with the AABB of 'that'
    double* const min = bdgBox->min;

```

```

double* const max = bdgBox->max;
const double* const thatBdgBoxMin = that->bdgBox.min;
const double* const thatBdgBoxMax = that->bdgBox.max;
for (int iAxis = 2;
     iAxis--;) {

    if (min[iAxis] < thatBdgBoxMin[iAxis]) {

        min[iAxis] = thatBdgBoxMin[iAxis];

    }
    if (max[iAxis] > thatBdgBoxMax[iAxis]) {

        max[iAxis] = thatBdgBoxMax[iAxis];

    }

}

}

// If we've reached here the two Frames are intersecting
return true;

}

```

## 5.2.4 3D dynamic

### Header

```

#ifndef __FMB3D_H_
#define __FMB3D_H_

#include <stdbool.h>
#include "frame.h"

// ----- Functions declaration -----

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
bool FMBTestIntersection3DTime(
    const Frame3D* const that,
    const Frame3D* const tho,
    AABB3D* const bdgBox);

#endif

```

### Body

```

#include "fmb3dt.h"

```

```

// ----- Macros -----

// Return 1.0 if v is positive, -1.0 if v is negative, 0.0 else
#define sgn(v) (((0.0 < (v)) ? 1 : 0) - (((v) < 0.0) ? 1 : 0))

#define FST_VAR 0
#define SND_VAR 1
#define THD_VAR 2

#define EPSILON 0.0000001 //0.001

// ----- Functions declaration -----

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// Return false if the system becomes inconsistent during elimination,
// else return true
bool ElimVar3DTime(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[3],
    double* Yp,
    int* const nbRemainRows);

// Get the bounds of the 'iVar'-th variable in the 'nbRows' rows
// system 'M'.X<='Y' and store them in the 'iVar'-th axis of the
// AABB 'bdgBox'
// ('M' arrangement is [iRow][iCol])
// The system is supposed to have been reduced to only one variable
// per row, the one in argument, which can be located in a different
// column than 'iVar'
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBound3D(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    AABB3D* const bdgBox);

// ----- Functions implementation -----

void PrintMY3DTime(
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbVar) {
    for (int iRow = 0; iRow < nbRows; ++iRow) {
        for (int iCol = 0; iCol < nbVar; ++iCol) {
            printf("%f ", M[iRow][iCol]);
        }
        printf("| %f\n", Y[iRow]);
    }
}

void PrintM3DTime(
    const double (*M)[3],

```



```

    const int nbRows) {
    for (int iRow = 0; iRow < nbRows; ++iRow) {
        for (int iCol = 0; iCol < 3; ++iCol) {
            printf("%f ", M[iRow][iCol]);
        }
        printf("\n");
    }
}

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// ('M' arrangement is [iRow][iCol])
// Return true if the system becomes inconsistent during elimination,
// else return false
bool ElimVar3DTime(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[3],
    double* Yp,
    int* const nbRemainRows) {

    // Initialize the number of rows in the result system
    *nbRemainRows = 0;

    // First we process the rows where the eliminated variable is not null

    // For each row except the last one
    for (int iRow = 0;
        iRow < nbRows - 1;
        ++iRow) {

        // Shortcuts
        int sgnMIRowIVar = sgn(M[iRow][iVar]);
        double fabsMIRowIVar = fabs(M[iRow][iVar]);
        double YIRowDivideByFabsMIRowIVar = Y[iRow] / fabsMIRowIVar;

        // For each following rows
        for (int jRow = iRow + 1;
            jRow < nbRows;
            ++jRow) {

            // If coefficients of the eliminated variable in the two rows have
            // different signs and are not null
            if (sgnMIRowIVar != sgn(M[jRow][iVar]) &&
                fabsMIRowIVar > EPSILON &&
                fabs(M[jRow][iVar]) > EPSILON) {

                // Declare a variable to memorize the sum of the negative
                // coefficients in the row
                double sumNegCoeff = 0.0;

                // Declare a variable to memorize if all the coefficients
                // are >= 0.0
                bool allPositive = true;

                // Declare a variable to memorize if all the coefficients
                // are null

```

```

bool allNull = true;

// Add the sum of the two normed (relative to the eliminated
// variable) rows into the result system. This actually
// eliminate the variable while keeping the constraints on
// others variables
for (int iCol = 0, jCol = 0;
    iCol < nbCols;
    ++iCol ) {

    if (iCol != iVar) {

        Mp[*nbRemainRows][jCol] =
            M[iRow][iCol] / fabsMIRowIVar +
            M[jRow][iCol] / fabs(M[jRow][iVar]);

        // If the coefficient is negative
        if (Mp[*nbRemainRows][jCol] < -1.0 * EPSILON) {

            // Memorize that at least one coefficient is not positive
            allPositive = false;

            // Memorize that at least one coefficient is not null
            allNull = false;

            // Update the sum of the negative coefficient
            sumNegCoeff += Mp[*nbRemainRows][jCol];

            // Else, if the coefficient is positive
        } else if (Mp[*nbRemainRows][jCol] > EPSILON) {

            // Memorize that at least one coefficient is not null
            allNull = false;

        }

        ++jCol;

    }

}

Yp[*nbRemainRows] =
    YIRowDivideByFabsMIRowIVar +
    Y[jRow] / fabs(M[jRow][iVar]);

// If at least one coefficient is not null
if (allNull == false) {

    // If all the coefficients are positive and the right side of
    // the inequality is negative
    if (allPositive == true &&
        Yp[*nbRemainRows] < 0.0) {

        // As X is in [0,1], the system is inconsistent
        return true;

    }

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Yp[*nbRemainRows] < sumNegCoeff) {

```

```

        // As X is in [0,1], the system is inconsistent
        return true;

    }

    // Else all coefficients are null, if the right side is null
    // or negative
    } else if (Yp[*nbRemainRows] <= 0.0) {

        // The system is inconsistent
        return true;

    }

    // Increment the nb of rows into the result system
    ++(*nbRemainRows);

}

}

}

// Then we copy and compress the rows where the eliminated
// variable is null

// Loop on rows of the input system
for (int iRow = 0;
     iRow < nbRows;
     ++iRow) {

    // Shortcut
    const double* MiRow = M[iRow];

    // If the coefficient of the eliminated variable is null on
    // this row
    if (fabs(M[iRow][iVar]) < EPSILON) {

        // Shortcut
        double* MpnbRemainRows = Mp[*nbRemainRows];

        // Copy this row into the result system excluding the eliminated
        // variable
        for (int iCol = 0, jCol = 0;
             iCol < nbCols;
             ++iCol) {

            if (iCol != iVar) {

                MpnbRemainRows[jCol] = MiRow[iCol];

                ++jCol;

            }

        }

        Yp[*nbRemainRows] = Y[iRow];

        // Increment the nb of rows into the result system
        ++(*nbRemainRows);
    }
}

```

```

    }

}

return false;

}

// Get the bounds of the 'iVar'-th variable in the 'nbRows' rows
// system 'M'.X<='Y' and store them in the 'iVar'-th axis of the
// AABB 'bdgBox'
// ('M' arrangement is [iRow][iCol])
// The system is supposed to have been reduced to only one variable
// per row, the one in argument
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBound3DTime(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    AABB3D* const bdgBox) {

    // Shortcuts
    double* min = bdgBox->min + iVar;
    double* max = bdgBox->max + iVar;

    // Initialize the bounds to there maximum maximum and minimum minimum
    *min = 0.0;
    *max = 1.0;

    // Loop on rows
    for (int jRow = 0;
        jRow < nbRows;
        ++jRow) {

        // Shortcut
        double MjRowiVar = M[jRow][0];

        // If this row has been reduced to the variable in argument
        // and it has a strictly positive coefficient
        if (MjRowiVar > EPSILON) {

            // Get the scaled value of Y for this row
            double y = Y[jRow] / MjRowiVar;

            // If the value is lower than the current maximum bound
            if (*max > y) {

                // Update the maximum bound
                *max = y;

            }

            // Else, if this row has been reduced to the variable in argument
            // and it has a strictly negative coefficient
        } else if (MjRowiVar < -1.0 * EPSILON) {

            // Get the scaled value of Y for this row
            double y = Y[jRow] / MjRowiVar;

```

```

        // If the value is greater than the current minimum bound
        if (*min < y) {

            // Update the minimum bound
            *min = y;

        }

    }

}

}

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
bool FMBTestIntersection3DTime(
    const Frame3D* const that,
    const Frame3D* const tho,
    AABB3D* const bdgBox) {

    // Get the projection of the Frame 'tho' in Frame 'that' coordinates
    // system
    Frame3D thoProj;
    Frame3DImportFrame(that, tho, &thoProj);

    // Declare two variables to memorize the system to be solved  $M.X \leq Y$ 
    // (M arrangement is [iRow][iCol])
    double M[14][3];
    double Y[14];

    // Shortcuts
    double (*thoProjComp)[3] = thoProj.comp;
    double *thoProjOrig = thoProj.orig;

    // Variable to memorise the current row in the system
    int iRow = 0;

    // Shortcuts
    double* MIRow;

    // Constraints  $0 + C.X \geq 0.0$ 
    // and constraints  $x_i \geq 0.0$ 
    for (;
        iRow < 3;
        ++iRow) {

        // Shortcuts
        MIRow = M[iRow];
        double* MJRow = M[iRow + 3];

        // For each column of the system
        double sumNeg = 0.0;
        for (int iCol = 3;
            iCol--;) {

```

```

    MIRow[iCol] = -1.0 * thoProjComp[iCol][iRow];
    if (MIRow[iCol] < 0.0) {
        sumNeg += MIRow[iCol];
    }

    // If it's on the diagonal
    if (iRow == iCol) {

        MJRow[iCol] = -1.0;

    // Else it's not on the diagonal
    } else {

        MJRow[iCol] = 0.0;

    }
}
if (thoProjOrig[iRow] < sumNeg)
    return false;
Y[iRow] = thoProjOrig[iRow];
Y[iRow + 3] = 0.0;
}
iRow = 6;

if (that->type == FrameCuboid) {

    // Constraints 0 + C.X <= 1.0
    for (int jRow = 0;
        jRow < 3;
        ++jRow, ++iRow) {

        // Shortcuts
        MIRow = M[iRow];
        double* MJRow = M[jRow];

        // For each column of the system
        double sumNeg = 0.0;
        for (int iCol = 3;
            iCol--;) {

            MIRow[iCol] = -1.0 * MJRow[iCol];
            if (MIRow[iCol] < 0.0) {
                sumNeg += MIRow[iCol];
            }

        }

        Y[iRow] = 1.0 - thoProjOrig[jRow];
        if (Y[iRow] < sumNeg)
            return false;
    }

}

// Else, the first frame is a Tetrahedron
} else {

    // Declare a variable to memorize the sum of the negative
    // coefficients in the row
    double sumNegCoeff = 0.0;

    // Declare a variable to memorize if all the coefficients
    // are null
    bool allNull = true;

```

```

// Shortcut
double* MRow = M[iRow];

Y[iRow] = 1.0;

// For each column of the system
for (int iCol = 3;
     iCol--;) {

    MRow[iCol] = 0.0;

    // For each component
    for (int iAxis = 3;
         iAxis--;) {

        MRow[iCol] += thoProjComp[iCol][iAxis];

    }

    Y[iRow] -= thoProjOrig[iCol];

    // If the coefficient is negative
    if (MRow[iCol] < -1.0 * EPSILON) {

        // Memorize that at least one coefficient is not null
        allNull = false;

        // Update the sum of the negative coefficient
        sumNegCoeff += MRow[iCol];

    }

    // Else, if the coefficient is positive
    } else if (MRow[iCol] > EPSILON) {

        // Memorize that at least one coefficient is not null
        allNull = false;

    }

}

// If at least one coefficient is not null
if (allNull == false) {

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[iRow] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        // there is no intersection
        return false;

    }

}

// Else all coefficients are null, if the right side is null
// or negative
} else if (Y[iRow] <= 0.0) {

    // The system is inconsistent, there is no intersection
    return false;

}

```

```

    // Update the number of rows in the system
    ++iRow;

}

if (tho->type == FrameCuboid) {

    // Constraints  $x_i \leq 1.0$ 
    for (int jRow = 0;
        jRow < 3;
        ++jRow, ++iRow) {

        // Shortcuts
        MRow = M[iRow];

        // For each column of the system
        for (int iCol = 3;
            iCol--;) {
            // If it's on the diagonal
            if (jRow == iCol) {

                MRow[iCol] = 1.0;

            // Else it's not on the diagonal
            } else {

                MRow[iCol] = 0.0;

            }

        }

        Y[iRow] = 1.0;

    }

// Else, the second frame is a Tetrahedron
} else {

    // Declare a variable to memorize the sum of the negative
    // coefficients in the row
    double sumNegCoeff = 0.0;

    // Declare a variable to memorize if all the coefficients
    // are null
    bool allNull = true;

    // Shortcut
    double* MRow = M[iRow];

    // For each column of the system
    for (int iCol = 3;
        iCol--;) {

        MRow[iCol] = 1.0;

        // If the coefficient is negative
        if (MRow[iCol] < -1.0 * EPSILON) {

            // Memorize that at least one coefficient is not null
            allNull = false;


```



```

        // Update the sum of the negative coefficient
        sumNegCoeff += MRow[iCol];

    // Else, if the coefficient is positive
    } else if (MRow[iCol] > EPSILON) {

        // Memorize that at least one coefficient is not null
        allNull = false;

    }

}

Y[iRow] = 1.0;

// If at least one coefficient is not null
if (allNull == false) {

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[iRow] < sumNegCoeff) {

        // As X is in [0,1], the system is inconsistent
        // there is no intersection
        return false;

    }

    // Else all coefficients are null, if the right side is null
    // or negative
    } else if (Y[iRow] <= 0.0) {

        // The system is inconsistent, there is no intersection
        return false;

    }

    // Update the number of rows in the system
    ++iRow;

}

// Declare a variable to memorize the total number of rows in the
// system. It may vary depending on the type of Frames
int nbRows = iRow;

// Solve the system

// Declare a AABB to memorize the bounding box of the intersection
// in the coordinates system of that
AABB3D bdgBoxLocal;

// Declare variables to eliminate the first variable
// The number of rows is set conservatively, one may try to reduce
// them if needed
double Mp[64][3];
double Yp[64];
int nbRowsP;

// Eliminate the first variable
bool inconsistency =
    ElimVar3DTime(

```

```

        FST_VAR,
        M,
        Y,
        nbRows,
        3,
        Mp,
        Yp,
        &nbRowsP);

// If the system is inconsistent
if (inconsistency == true) {

    // The two Frames are not in intersection
    return false;

}

// Declare variables to eliminate the second variable
// The number of rows is set conservatively, one may try to reduce
// them if needed
double Mpp[514][3];
double Ypp[514];
int nbRowsPP;

// Eliminate the second variable (which is the first in the new system)
inconsistency =
    ElimVar3DTime(
        FST_VAR,
        Mp,
        Yp,
        nbRowsP,
        2,
        Mpp,
        Ypp,
        &nbRowsPP);

// If the system is inconsistent
if (inconsistency == true) {

    // The two Frames are not in intersection
    return false;

}

// Get the bounds for the remaining third variable
GetBound3DTime(
    THD_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

// If the bounds are inconstant
if (bdgBoxLocal.min[THD_VAR] >= bdgBoxLocal.max[THD_VAR]) {

    // The two Frames are not in intersection
    return false;

}

// Eliminate the third variable (which is second first in the new
// system)

```

```

inconsistency =
    ElimVar3DTime(
        SND_VAR,
        Mp,
        Yp,
        nbRowsP,
        2,
        Mpp,
        Ypp,
        &nbRowsPP);

// If the resulting system is inconsistent
if (inconsistency == true) {

    // The two Frames are not in intersection
    return false;

// Else, if the bounds are consistent here it means
// the two Frames are in intersection.
// If the user hasn't requested for the resulting bounding box
} else if (bdgBox == NULL) {

    // Immediately return true
    return true;

}

// Get the bounds for the remaining second variable
GetBound3DTime(
    SND_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

// If the bounds are inconstant
if (bdgBoxLocal.min[SND_VAR] >= bdgBoxLocal.max[SND_VAR]) {

    // The two Frames are not in intersection
    return false;

}

// Now starts again from the initial systems and eliminate the
// second and third variables to get the bounds of the first variable
inconsistency =
    ElimVar3DTime(
        SND_VAR,
        M,
        Y,
        nbRows,
        3,
        Mp,
        Yp,
        &nbRowsP);

if (inconsistency == true) {
    return false;
}

inconsistency =
    ElimVar3DTime(

```

```

        SND_VAR,
        Mp,
        Yp,
        nbRowsP,
        2,
        Mpp,
        Ypp,
        &nbRowsPP);

if (inconsistency == true) {
    return false;
}

GetBound3DTime(
    FST_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

if (bdgBoxLocal.min[FST_VAR] >= bdgBoxLocal.max[FST_VAR]) {
    return false;
}

// If the user requested the resulting bounding box
if (bdgBox != NULL) {

    // Export the local bounding box toward the real coordinates
    // system
    Frame3DExportBdgBox(
        tho,
        &bdgBoxLocal,
        bdgBox);

    // Clip with the AABB of 'that'
    double* const min = bdgBox->min;
    double* const max = bdgBox->max;
    const double* const thatBdgBoxMin = that->bdgBox.min;
    const double* const thatBdgBoxMax = that->bdgBox.max;
    for (int iAxis = 3;
        iAxis--;) {

        if (min[iAxis] < thatBdgBoxMin[iAxis]) {

            min[iAxis] = thatBdgBoxMin[iAxis];

        }
        if (max[iAxis] > thatBdgBoxMax[iAxis]) {

            max[iAxis] = thatBdgBoxMax[iAxis];

        }
    }
}

// If we've reached here the two Frames are intersecting
return true;
}

```

## 5.3 Example of use

```
// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

// Include FMB algorithm library
#include "fmb2d.h"
#include "fmb3d.h"

// Main function
int main(int argc, char** argv) {

    // ----- 3D -----

    // Create the two objects to be tested for intersection
    double origP3D[3] = {0.0, 0.0, 0.0};
    double compP3D[3][3] = {
        {1.0, 0.0, 0.0}, // First component
        {0.0, 1.0, 0.0}, // Second component
        {0.0, 0.0, 1.0}}; // Third component
    Frame3D P3D =
        Frame3DCreateStatic(
            FrameTetrahedron,
            origP3D,
            compP3D);

    double origQ3D[3] = {0.5, 0.5, 0.5};
    double compQ3D[3][3] = {
        {2.0, 0.0, 0.0},
        {0.0, 2.0, 0.0},
        {0.0, 0.0, 2.0}};
    Frame3D Q3D =
        Frame3DCreateStatic(
            FrameTetrahedron,
            origQ3D,
            compQ3D);

    // Declare a variable to memorize the result of the intersection
    // detection
    AAB3D bdgBox3D;

    // Test for intersection between P and Q
    bool isIntersecting3D =
        FMBTestIntersection3D(
            &P3D,
            &Q3D,
            &bdgBox3D);

    // If the two objects are intersecting
    if (isIntersecting3D) {

        printf("Intersection detected in AAB3D ");
        AAB3DPrint(&bdgBox3D);
        printf("\n");

    } else {

        // Else, the two objects are not intersecting
        printf("No intersection.\n");

    }

}
```

```

}

// ----- 2D -----

// Create the two objects to be tested for intersection
double origP2D[2] = {0.0, 0.0};
double compP2D[2][2] = {
    {1.0, 0.0}, // First component
    {0.0, 1.0}}; // Second component
Frame2D P2D =
    Frame2DCreateStatic(
        FrameTetrahedron,
        origP2D,
        compP2D);

double origQ2D[2] = {0.5, 0.5};
double compQ2D[2][2] = {
    {-1.0, 0.0},
    {0.0, -1.0}};
Frame2D Q2D =
    Frame2DCreateStatic(
        FrameTetrahedron,
        origQ2D,
        compQ2D);

// Declare a variable to memorize the result of the intersection
// detection
AABB2D bdgBox2D;

// Test for intersection between P and Q
bool isIntersecting2D =
    FMBTestIntersection2D(
        &P2D,
        &Q2D,
        &bdgBox2D);

// If the two objects are intersecting
if (isIntersecting2D) {

    printf("Intersection detected in AABB ");
    AABB2DPrint(&bdgBox2D);
    printf("\n");

// Else, the two objects are not intersecting
} else {

    printf("No intersection.\n");

}

return 0;
}

```

## 6 Validation

In this section I introduce the code I've used to validate the algorithm and its implementation. The validation consists of, first running the algorithm on a set of unit test for which the solution has been computed by hand, and second running the FMB algorithm on randomly generated pairs of Frame and check that its result is equal to the one of running the SAT algorithm on the same pair of Frames. The code of the implementation of the SAT algorithm is given in annex (p.119)

### 6.1 Code

#### 6.1.1 Unit tests

```
// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

// Include FMB algorithm library
#include "fmb2d.h"
#include "fmb3d.h"

// Epsilon for numerical precision
#define EPSILON 0.0001

// Helper structure to pass arguments to the UnitTest function
typedef struct {
    FrameType type;
    double orig[2];
    double comp[2][2];
} Param2D;
typedef struct {
    FrameType type;
    double orig[3];
    double comp[3][3];
} Param3D;

// Unit test function
// Takes two Frame definitions, the correct answer in term of
// intersection/no intersection and the correct bounding box
// Run the FMB intersection detection algorithm on the Frames
// and check against the correct results
void UnitTest2D(
    const Param2D paramP,
    const Param2D paramQ,
    const bool correctAnswer,
    const AABB2D* const correctBdgBox) {

    // Create the two Frames
    Frame2D P =
        Frame2DCreateStatic(
            paramP.type,
            paramP.orig,
```

```

    paramP.comp);

Frame2D Q =
    Frame2DCreateStatic(
        paramQ.type,
        paramQ.orig,
        paramQ.comp);

// Declare a variable to memorize the resulting bounding box
AABB2D bdgBox;

// Helper variables to loop on the pair (that, tho) and (tho, that)
Frame2D* that = &P;
Frame2D* tho = &Q;

// Loop on pairs of Frames
for (int iPair = 2;
     iPair--;) {

    // Display the tested frames
    Frame2DPrint(that);
    printf("\nagainst\n");
    Frame2DPrint(tho);
    printf("\n");

    // Run the FMB intersection test
    bool isIntersecting =
        FMBTestIntersection2D(
            that,
            tho,
            &bdgBox);

    // If the test hasn't given the expected answer about intersection
    if (isIntersecting != correctAnswer) {

        // Display information about the failure
        printf(" Failed\n");
        printf("Expected : ");
        if (correctAnswer == false)
            printf("no ");
        printf("intersection\n");
        printf("Got : ");
        if (isIntersecting == false)
            printf("no ");
        printf("intersection\n");
        exit(0);

    // Else, the test has given the expected answer about intersection
    } else {

        // If the Frames were intersecting
        if (isIntersecting == true) {

            // Check the bounding box
            bool flag = true;
            for (int i = 2;
                 i--;) {

                if (bdgBox.min[i] > correctBdgBox->min[i] + EPSILON ||
                    bdgBox.max[i] < correctBdgBox->max[i] - EPSILON) {

                    flag = false;

```



```

    }

}

// If the bounding box is the expected one
if (flag == true) {

    // Display information
    printf("Succeed\n");

// Else, the bounding box wasn't the expected one
} else {

    // Display information
    printf("Failed\n");
    printf("Expected : ");
    AABB2DPrint(correctBdgBox);
    printf("\n");
    printf("    Got : ");
    AABB2DPrint(&bdgBox);
    printf("\n");

    // Terminate the unit tests
    exit(0);

}

// Else the Frames were not intersected,
// no need to check the bounding box
} else {

    // Display information
    printf(" Succeed\n");

}

}

printf("\n");

// Flip the pair of Frames
that = &Q;
tho = &P;

}

}

void UnitTest3D(
    const Param3D paramP,
    const Param3D paramQ,
    const bool correctAnswer,
    const AABB3D* const correctBdgBox) {

    // Create the two Frames
    Frame3D P =
        Frame3DCreateStatic(
            paramP.type,
            paramP.orig,
            paramP.comp);

    Frame3D Q =

```

```

Frame3DCreateStatic(
    paramQ.type,
    paramQ.orig,
    paramQ.comp);

// Declare a variable to memorize the resulting bounding box
AABB3D bdgBox;

// Helper variables to loop on the pair (that, tho) and (tho, that)
Frame3D* that = &P;
Frame3D* tho = &Q;

// Loop on pairs of Frames
for (int iPair = 2;
    iPair--;) {

    // Display the tested frames
    Frame3DPrint(that);
    printf("\nagainst\n");
    Frame3DPrint(tho);
    printf("\n");

    // Run the FMB intersection test
    bool isIntersecting =
        FMBTestIntersection3D(
            that,
            tho,
            &bdgBox);

    // If the test hasn't given the expected answer about intersection
    if (isIntersecting != correctAnswer) {

        // Display information about the failure
        printf(" Failed\n");
        printf("Expected : ");
        if (correctAnswer == false)
            printf("no ");
        printf("intersection\n");
        printf("Got : ");
        if (isIntersecting == false)
            printf("no ");
        printf("intersection\n");
        exit(0);

    // Else, the test has given the expected answer about intersection
    } else {

        // If the Frames were intersecting
        if (isIntersecting == true) {

            // Check the bounding box
            bool flag = true;
            for (int i = 3;
                i--;) {

                if (bdgBox.min[i] > correctBdgBox->min[i] + EPSILON ||
                    bdgBox.max[i] < correctBdgBox->max[i] - EPSILON) {

                    flag = false;
                }
            }
        }
    }
}

```

```

    }

    // If the bounding box is the expected one
    if (flag == true) {

        // Display information
        printf("Succeed\n");

    // Else, the bounding box wasn't the expected one
    } else {

        // Display information
        printf("Failed\n");
        printf("Expected : ");
        AABB3DPrint(correctBdgBox);
        printf("\n");
        printf("      Got : ");
        AABB3DPrint(&bdgBox);
        printf("\n");

        // Terminate the unit tests
        exit(0);

    }

    // Else the Frames were not intersected,
    // no need to check the bounding box
    } else {

        // Display information
        printf(" Succeed\n");

    }

}

printf("\n");

// Flip the pair of Frames
that = &Q;
tho = &P;

}

}

// Main function
void Test3D(void) {

    // Declare two variables to memoize the arguments to the
    // Validation function
    Param3D paramP;
    Param3D paramQ;

    // Declare a variable to memorize the correct bounding box
    AABB3D correctBdgBox;

    // Execute the unit test on various cases

    // -----
    paramP = (Param3D)
    { .type = FrameCuboid,
      .orig = {0.0, 0.0, 0.0},

```

```

        .comp =
            {{1.0, 0.0, 0.0},
             {0.0, 1.0, 0.0},
             {0.0, 0.0, 1.0}}
    };
    paramQ = (Param3D)
    {
        .type = FrameCuboid,
        .orig = {0.0, 0.0, 0.0},
        .comp =
            {{1.0, 0.0, 0.0},
             {0.0, 1.0, 0.0},
             {0.0, 0.0, 1.0}}
    };
    correctBdgBox = (AABB3D)
    {
        .min = {0.0, 0.0, 0.0},
        .max = {1.0, 1.0, 1.0}
    };
    UnitTest3D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
    paramP = (Param3D)
    {
        .type = FrameCuboid,
        .orig = {0.0, 0.0, 0.0},
        .comp =
            {{1.0, 0.0, 0.0},
             {0.0, 1.0, 0.0},
             {0.0, 0.0, 1.0}}
    };
    paramQ = (Param3D)
    {
        .type = FrameCuboid,
        .orig = {0.5, 0.5, 0.5},
        .comp =
            {{1.0, 0.0, 0.0},
             {0.0, 1.0, 0.0},
             {0.0, 0.0, 1.0}}
    };
    correctBdgBox = (AABB3D)
    {
        .min = {0.5, 0.5, 0.5},
        .max = {1.0, 1.0, 1.0}
    };
    UnitTest3D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
    paramP = (Param3D)
    {
        .type = FrameCuboid,
        .orig = {0.0, 0.0, 0.0},
        .comp =
            {{1.0, 0.0, 0.0},
             {0.0, 1.0, 0.0},
             {0.0, 0.0, 1.0}}
    };
    paramQ = (Param3D)
    {
        .type = FrameCuboid,
        .orig = {-0.5, -0.5, -0.5},

```

```

        .comp =
            {{1.0, 0.0, 0.0},
             {0.0, 1.0, 0.0},
             {0.0, 0.0, 1.0}}
    };
    correctBdgBox = (AABB3D)
        {.min = {0.0, 0.0, 0.0},
         .max = {0.5, 0.5, 0.5}}
    };
    UnitTest3D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
    paramP = (Param3D)
        {.type = FrameCuboid,
         .orig = {0.0, 0.0, 0.0},
         .comp =
             {{1.0, 0.0, 0.0},
              {0.0, 1.0, 0.0},
              {0.0, 0.0, 1.0}}
        };
    paramQ = (Param3D)
        {.type = FrameCuboid,
         .orig = {1.5, 1.5, 1.5},
         .comp =
             {{-1.0, 0.0, 0.0},
              {0.0, -1.0, 0.0},
              {0.0, 0.0, -1.0}}
        };
    correctBdgBox = (AABB3D)
        {.min = {0.5, 0.5, 0.5},
         .max = {1.0, 1.0, 1.0}}
    };
    UnitTest3D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
    paramP = (Param3D)
        {.type = FrameCuboid,
         .orig = {0.0, 0.0, 0.0},
         .comp =
             {{1.0, 0.0, 0.0},
              {0.0, 1.0, 0.0},
              {0.0, 0.0, 1.0}}
        };
    paramQ = (Param3D)
        {.type = FrameCuboid,
         .orig = {0.5, 1.5, -1.5},
         .comp =
             {{1.0, 0.0, 0.0},
              {0.0, -1.0, 0.0},
              {0.0, 0.0, 1.0}}
        };
    UnitTest3D(
        paramP,
        paramQ,

```

```

        false,
        NULL);

// -----
paramP = (Param3D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0, 0.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {0.0, 1.0, 0.0},
     {0.0, 0.0, -1.0}}
};
paramQ = (Param3D)
{.type = FrameCuboid,
 .orig = {0.5, 1.5, -1.5},
 .comp =
    {{1.0, 0.0, 0.0},
     {0.0, -1.0, 0.0},
     {0.0, 0.0, 1.0}}
};
correctBdgBox = (AABB3D)
{.min = {0.5, 0.5, -1.0},
 .max = {1.0, 1.0, -0.5}
};
UnitTest3D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param3D)
{.type = FrameCuboid,
 .orig = {-1.0, -1.0, -1.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {1.0, 1.0, 1.0},
     {0.0, 0.0, 1.0}}
};
paramQ = (Param3D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0, 0.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {0.0, 1.0, 0.0},
     {0.0, 0.0, 1.0}}
};
UnitTest3D(
    paramP,
    paramQ,
    false,
    NULL);

// -----
paramP = (Param3D)
{.type = FrameCuboid,
 .orig = {-1.0, -1.0, -1.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {1.0, 1.0, 1.0},
     {0.0, 0.0, 1.0}}
};

```

```

paramQ = (Param3D)
{.type = FrameCuboid,
 .orig = {0.0, -0.5, 0.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {0.0, 1.0, 0.0},
     {0.0, 0.0, 1.0}}
};
correctBdgBox = (AABB3D)
{.min = {0.0, -0.5, 0.0},
 .max = {1.0, 0.0, 1.0}
};
UnitTest3D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param3D)
{.type = FrameTetrahedron,
 .orig = {-1.0, -1.0, -1.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {1.0, 1.0, 1.0},
     {0.0, 0.0, 1.0}}
};
paramQ = (Param3D)
{.type = FrameCuboid,
 .orig = {0.0, -0.5, 0.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {0.0, 1.0, 0.0},
     {0.0, 0.0, 1.0}}
};
UnitTest3D(
    paramP,
    paramQ,
    false,
    NULL);

// -----
paramP = (Param3D)
{.type = FrameCuboid,
 .orig = {-1.0, -1.0, -1.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {1.0, 1.0, 1.0},
     {0.0, 0.0, 1.0}}
};
paramQ = (Param3D)
{.type = FrameTetrahedron,
 .orig = {0.0, -0.5, 0.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {0.0, 1.0, 0.0},
     {0.0, 0.0, 1.0}}
};
correctBdgBox = (AABB3D)
{.min = {0.0, -0.5, 0.0},
 .max = {0.75, 0.0, 0.75}
};

```

```

UnitTest3D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param3D)
{.type = FrameTetrahedron,
 .orig = {-1.0, -1.0, -1.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {1.0, 1.0, 1.0},
     {0.0, 0.0, 1.0}}
};
paramQ = (Param3D)
{.type = FrameTetrahedron,
 .orig = {0.0, -0.5, 0.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {0.0, 1.0, 0.0},
     {0.0, 0.0, 1.0}}
};
UnitTest3D(
    paramP,
    paramQ,
    false,
    NULL);

// -----
paramP = (Param3D)
{.type = FrameTetrahedron,
 .orig = {-0.5, -1.0, -0.5},
 .comp =
    {{1.0, 0.0, 0.0},
     {1.0, 1.0, 1.0},
     {0.0, 0.0, 1.0}}
};
paramQ = (Param3D)
{.type = FrameTetrahedron,
 .orig = {0.0, -0.5, 0.0},
 .comp =
    {{1.0, 0.0, 0.0},
     {0.0, 1.0, 0.0},
     {0.0, 0.0, 1.0}}
};
correctBdgBox = (AABB3D)
{.min = {0.0, -0.5, 0.0},
 .max = {0.5, -0.5 + 1.0 / 3.0, 0.5}
};
UnitTest3D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// If we reached here, it means all the unit tests succeed
printf("All unit tests 3D have succeed.\n");
}

```



```

void Test2D(void) {

    // Declare two variables to memoize the arguments to the
    // Validation function
    Param2D paramP;
    Param2D paramQ;

    // Declare a variable to memorize the correct bounding box
    AABB2D correctBdgBox;

    // Execute the unit test on various cases

    // -----
    paramP = (Param2D)
    {
        .type = FrameCuboid,
        .orig = {0.0, 0.0},
        .comp =
        {
            {1.0, 0.0},
            {0.0, 1.0}
        }
    };
    paramQ = (Param2D)
    {
        .type = FrameCuboid,
        .orig = {0.0, 0.0},
        .comp =
        {
            {1.0, 0.0},
            {0.0, 1.0}
        }
    };
    correctBdgBox = (AABB2D)
    {
        .min = {0.0, 0.0},
        .max = {1.0, 1.0}
    };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

    // -----
    paramP = (Param2D)
    {
        .type = FrameCuboid,
        .orig = {0.0, 0.0},
        .comp =
        {
            {1.0, 0.0},
            {0.0, 1.0}
        }
    };
    paramQ = (Param2D)
    {
        .type = FrameCuboid,
        .orig = {0.5, 0.5},
        .comp =
        {
            {1.0, 0.0},
            {0.0, 1.0}
        }
    };
    correctBdgBox = (AABB2D)
    {
        .min = {0.5, 0.5},
        .max = {1.0, 1.0}
    };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);
}

```

```

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {-0.5, -0.5},
 .comp =
   {{1.0, 0.0},
    {0.0, 1.0}}
};
paramQ = (Param2D)
{.type = FrameCuboid,
 .orig = {0.5, 0.5},
 .comp =
   {{1.0, 0.0},
    {0.0, 1.0}}
};
UnitTest2D(
  paramP,
  paramQ,
  false,
  NULL);

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0},
 .comp =
   {{1.0, 0.0},
    {0.0, 1.0}}
};
paramQ = (Param2D)
{.type = FrameCuboid,
 .orig = {0.25, -0.25},
 .comp =
   {{0.5, 0.0},
    {0.0, 2.0}}
};
correctBdgBox = (AABB2D)
{.min = {0.25, 0.0},
 .max = {0.75, 1.0}
};
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0},
 .comp =
   {{1.0, 0.0},
    {0.0, 1.0}}
};
paramQ = (Param2D)
{.type = FrameCuboid,
 .orig = {-0.25, 0.25},
 .comp =
   {{2.0, 0.0},
    {0.0, 0.5}}
};
correctBdgBox = (AABB2D)

```

```

        {.min = {0.0, 0.25},
         .max = {1.0, 0.75}
        };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
    paramP = (Param2D)
        {.type = FrameCuboid,
         .orig = {0.0, 0.0},
         .comp =
             {{1.0, 1.0},
              {-1.0, 1.0}}
        };
    paramQ = (Param2D)
        {.type = FrameCuboid,
         .orig = {0.0, 0.0},
         .comp =
             {{1.0, 0.0},
              {0.0, 1.0}}
        };
    correctBdgBox = (AABB2D)
        {.min = {0.0, 0.0},
         .max = {1.0, 1.0}
        };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
    paramP = (Param2D)
        {.type = FrameCuboid,
         .orig = {-0.5, -0.5},
         .comp =
             {{1.0, 1.0},
              {-1.0, 1.0}}
        };
    paramQ = (Param2D)
        {.type = FrameCuboid,
         .orig = {0.0, 0.0},
         .comp =
             {{1.0, 0.0},
              {0.0, 1.0}}
        };
    correctBdgBox = (AABB2D)
        {.min = {0.0, 0.0},
         .max = {0.5, 1.0}
        };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
    paramP = (Param2D)
        {.type = FrameCuboid,

```

```

        .orig = {1.5, 1.5},
        .comp =
            {{1.0, -1.0},
             {-1.0, -1.0}}
    };
    paramQ = (Param2D)
    {
        .type = FrameCuboid,
        .orig = {1.0, 0.0},
        .comp =
            {{-1.0, 0.0},
             {0.0, 1.0}}
    };
    correctBdgBox = (AABB2D)
    {
        .min = {0.5, 0.0},
        .max = {1.0, 1.0}
    };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
    paramP = (Param2D)
    {
        .type = FrameCuboid,
        .orig = {1.0, 0.5},
        .comp =
            {{-0.5, 0.5},
             {-0.5, -0.5}}
    };
    paramQ = (Param2D)
    {
        .type = FrameCuboid,
        .orig = {0.0, 1.0},
        .comp =
            {{1.0, 0.0},
             {0.0, -1.0}}
    };
    correctBdgBox = (AABB2D)
    {
        .min = {0.0, 0.0},
        .max = {1.0, 1.0}
    };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
    paramP = (Param2D)
    {
        .type = FrameCuboid,
        .orig = {0.0, 0.0},
        .comp =
            {{1.0, 0.0},
             {1.0, 1.0}}
    };
    paramQ = (Param2D)
    {
        .type = FrameCuboid,
        .orig = {2.0, -1.0},
        .comp =
            {{0.0, 1.0},
             {-0.5, 1.0}}
    };

```

```

correctBdgBox = (AABB2D)
    {.min = {1.5, 0.5},
     .max = {1.5 + 0.5 / 3.0, 1.0}
    };
UnitTest2D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param2D)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0},
     .comp =
         {{1.0, 0.5},
          {0.5, 1.0}}
    };
paramQ = (Param2D)
    {.type = FrameCuboid,
     .orig = {1.0, 1.0},
     .comp =
         {{-0.5, -0.5},
          {0.0, -1.0}}
    };
correctBdgBox = (AABB2D)
    {.min = {0.5, 0.25},
     .max = {1.0, 1.0}
    };
UnitTest2D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param2D)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0},
     .comp =
         {{1.0, 0.5},
          {0.5, 1.0}}
    };
paramQ = (Param2D)
    {.type = FrameCuboid,
     .orig = {1.0, 2.0},
     .comp =
         {{-0.5, -0.5},
          {0.0, -1.0}}
    };
correctBdgBox = (AABB2D)
    {.min = {0.5, 0.75},
     .max = {1.0, 1.25}
    };
UnitTest2D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param2D)

```

```

        {.type = FrameTetrahedron,
         .orig = {0.0, 0.0},
         .comp =
             {{1.0, 0.5},
              {0.5, 1.0}}
        };
paramQ = (Param2D)
{.type = FrameCuboid,
 .orig = {1.0, 2.0},
 .comp =
     {{-0.5, -0.5},
      {0.0, -1.0}}
};
correctBdgBox = (AABB2D)
{.min = {0.5, 0.5},
 .max = {0.75, 1.0}
};
UnitTest2D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0},
 .comp =
     {{1.0, 0.5},
      {0.5, 1.0}}
};
paramQ = (Param2D)
{.type = FrameTetrahedron,
 .orig = {1.0, 2.0},
 .comp =
     {{-0.5, -0.5},
      {0.0, -1.0}}
};
correctBdgBox = (AABB2D)
{.min = {0.5 + 1.0 / 3.0, 1.0},
 .max = {1.0, 1.0 + 1.0 / 3.0}
};
UnitTest2D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0},
 .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
};
paramQ = (Param2D)
{.type = FrameTetrahedron,
 .orig = {0.0, 0.0},
 .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
};

```

```

    };
    correctBdgBox = (AABB2D)
        {.min = {0.0, 0.0},
         .max = {1.0, 1.0}
        };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

// -----
paramP = (Param2D)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0},
     .comp =
         {{1.0, 0.0},
          {0.0, 1.0}}
    };
paramQ = (Param2D)
    {.type = FrameTetrahedron,
     .orig = {0.0, -0.5},
     .comp =
         {{1.0, 0.0},
          {0.0, 1.0}}
    };
correctBdgBox = (AABB2D)
    {.min = {0.0, 0.0},
     .max = {0.5, 0.5}
    };
UnitTest2D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param2D)
    {.type = FrameCuboid,
     .orig = {0.5, 0.5},
     .comp =
         {{-0.5, 0.0},
          {0.0, -0.5}}
    };
paramQ = (Param2D)
    {.type = FrameTetrahedron,
     .orig = {0.0, -0.5},
     .comp =
         {{1.0, 0.0},
          {0.0, 1.0}}
    };
correctBdgBox = (AABB2D)
    {.min = {0.0, 0.0},
     .max = {0.5, 0.5}
    };
UnitTest2D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----

```

```

paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.5, 0.5},
 .comp =
  {{1.0, 0.0},
   {0.0, 1.0}}
};
paramQ = (Param2D)
{.type = FrameTetrahedron,
 .orig = {0.0, 0.0},
 .comp =
  {{1.0, 0.0},
   {0.0, 1.0}}
};
UnitTest2D(
  paramP,
  paramQ,
  false,
  NULL);

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0},
 .comp =
  {{1.0, 0.0},
   {0.0, 1.0}}
};
paramQ = (Param2D)
{.type = FrameTetrahedron,
 .orig = {1.5, 1.5},
 .comp =
  {{-1.5, 0.0},
   {0.0, -1.5}}
};
correctBdgBox = (AABB2D)
{.min = {0.5, 0.5},
 .max = {1.0, 1.0}
};
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// -----
paramP = (Param2D)
{.type = FrameTetrahedron,
 .orig = {0.0, 0.0},
 .comp =
  {{1.0, 0.0},
   {0.0, 1.0}}
};
paramQ = (Param2D)
{.type = FrameTetrahedron,
 .orig = {1.0, 1.0},
 .comp =
  {{-1.0, 0.0},
   {0.0, -1.0}}
};
UnitTest2D(
  paramP,

```



```

    paramQ,
    false,
    NULL);

// -----
paramP = (Param2D)
{.type = FrameTetrahedron,
 .orig = {0.0, 0.0},
 .comp =
    {{1.0, 0.5},
     {0.5, 1.0}}
};
paramQ = (Param2D)
{.type = FrameTetrahedron,
 .orig = {1.0, 1.0},
 .comp =
    {{-0.5, -0.5},
     {0.0, -1.0}}
};
correctBdgBox = (AABB2D)
{.min = {0.5, 0.5 - 1.0 / 6.0},
 .max = {1.0, 0.75}
};
UnitTest2D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

// -----
paramP = (Param2D)
{.type = FrameTetrahedron,
 .orig = {0.0, 0.0},
 .comp =
    {{1.0, 0.5},
     {0.5, 1.0}}
};
paramQ = (Param2D)
{.type = FrameTetrahedron,
 .orig = {1.0, 1.5},
 .comp =
    {{-0.5, -0.5},
     {0.0, -1.0}}
};
UnitTest2D(
    paramP,
    paramQ,
    false,
    NULL);

// If we reached here, it means all the unit tests succeed
printf("All unit tests 2D have succeed.\n");

}

// Main function
int main(int argc, char** argv) {

    Test2D();
    Test3D();

    return 0;
}

```

```
}
```

## 6.1.2 Validation against SAT

```
// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>

// Include FMB and SAT algorithm library
#include "fmb2d.h"
#include "fmb3d.h"
#include "sat.h"

// Epsilon for numerical precision
#define EPSILON 0.0001
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

// Helper structure to pass arguments to the Validation function
typedef struct {
    FrameType type;
    double orig[2];
    double comp[2][2];
} Param2D;
typedef struct {
    FrameType type;
    double orig[3];
    double comp[3][3];
} Param3D;

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void Validation2D(
    const Param2D paramP,
    const Param2D paramQ) {

    // Create the two Frames
    Frame2D P =
        Frame2DCreateStatic(
            paramP.type,
            paramP.orig,
            paramP.comp);

    Frame2D Q =
        Frame2DCreateStatic(
            paramQ.type,
            paramQ.orig,
            paramQ.comp);
```

```

// Helper variables to loop on the pair (that, tho) and (tho, that)
Frame2D* that = &P;
Frame2D* tho = &Q;

// Loop on pairs of Frames
for (int iPair = 2;
     iPair--;) {

    // Test intersection with FMB
    bool isIntersectingFMB =
        FMBTestIntersection2D(
            that,
            tho,
            NULL);

    // Test intersection with SAT
    bool isIntersectingSAT =
        SATTestIntersection2D(
            that,
            tho);

    // If the results are different
    if (isIntersectingFMB != isIntersectingSAT) {

        // Print the disagreement
        printf("Validation2D has failed\n");
        Frame2DPrint(that);
        printf(" against ");
        Frame2DPrint(tho);
        printf("\n");
        printf("FMB : ");
        if (isIntersectingFMB == false)
            printf("no ");
        printf("intersection\n");
        printf("SAT : ");
        if (isIntersectingSAT == false)
            printf("no ");
        printf("intersection\n");

        // Stop the validation
        exit(0);
    }

    // If the Frames are in intersection
    if (isIntersectingFMB == true) {

        // Update the number of intersection
        nbInter++;

        // If the Frames are not in intersection
    } else {

        // Update the number of no intersection
        nbNoInter++;
    }

    // Flip the pair of Frames
    that = &Q;
    tho = &P;
}

```

```

    }

}

void Validation3D(
    const Param3D paramP,
    const Param3D paramQ) {

    // Create the two Frames
    Frame3D P =
        Frame3DCreateStatic(
            paramP.type,
            paramP.orig,
            paramP.comp);

    Frame3D Q =
        Frame3DCreateStatic(
            paramQ.type,
            paramQ.orig,
            paramQ.comp);

    // Helper variables to loop on the pair (that, tho) and (tho, that)
    Frame3D* that = &P;
    Frame3D* tho = &Q;

    // Loop on pairs of Frames
    for (int iPair = 2;
        iPair--;) {

        // Test intersection with FMB
        bool isIntersectingFMB =
            FMBTestIntersection3D(
                that,
                tho,
                NULL);

        // Test intersection with SAT
        bool isIntersectingSAT =
            SATTestIntersection3D(
                that,
                tho);

        // If the results are different
        if (isIntersectingFMB != isIntersectingSAT) {

            // Print the disagreement
            printf("Validation3D has failed\n");
            Frame3DPrint(that);
            printf(" against ");
            Frame3DPrint(tho);
            printf("\n");
            printf("FMB : ");
            if (isIntersectingFMB == false)
                printf("no ");
            printf("intersection\n");
            printf("SAT : ");
            if (isIntersectingSAT == false)
                printf("no ");
            printf("intersection\n");

            // Stop the validation

```

```

        exit(0);

    }

    // If the Frames are in intersection
    if (isIntersectingFMB == true) {

        // Update the number of intersection
        nbInter++;

    // If the Frames are not in intersection
    } else {

        // Update the number of no intersection
        nbNoInter++;

    }

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

}

}

// Main function
void Validate2D(void) {

    // Initialise the random generator
    srand(time(NULL));

    // Declare two variables to memorize the arguments to the
    // Validation function
    Param2D paramP;
    Param2D paramQ;

    // Initialize the number of intersection and no intersection
    nbInter = 0;
    nbNoInter = 0;

    // Loop on the tests
    for (unsigned long iTest = NB_TESTS;
        iTest--;) {

        // Create two random Frame definitions
        Param2D* param = &paramP;
        for (int iParam = 2;
            iParam--;) {

            // 50% chance of being a Cuboid or a Tetrahedron
            if (rnd() < 0.5)
                param->type = FrameCuboid;
            else
                param->type = FrameTetrahedron;

            for (int iAxis = 2;
                iAxis--;) {

                param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

                for (int iComp = 2;

```

```

        iComp--;) {

            param->comp[iComp][iAxis] =
                -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

        }

    }

    param = &paramQ;

}

// Calculate the determinant of the Frames' components matrix
double detP =
    paramP.comp[0][0] * paramP.comp[1][1] -
    paramP.comp[1][0] * paramP.comp[0][1];

double detQ =
    paramQ.comp[0][0] * paramQ.comp[1][1] -
    paramQ.comp[1][0] * paramQ.comp[0][1];

// If the determinants are not null, ie the Frame are not degenerate
if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Validation2D(
        paramP,
        paramQ);

}

}

// If we reached it means the validation was successfull
// Print results
printf("Validation2D has succeed.\n");
printf("Tested %lu intersections ", nbInter);
printf("and %lu no intersections\n", nbNoInter);

}

void Validate3D(void) {

    // Initialise the random generator
    srand(time(NULL));

    // Declare two variables to memorize the arguments to the
    // Validation function
    Param3D paramP;
    Param3D paramQ;

    // Initialize the number of intersection and no intersection
    nbInter = 0;
    nbNoInter = 0;

    // Loop on the tests
    for (unsigned long iTest = NB_TESTS;
        iTest--;) {

        // Create two random Frame definitions
        Param3D* param = &paramP;

```

```

for (int iParam = 2;
    iParam--;) {

    // 50% chance of being a Cuboid or a Tetrahedron
    if (rnd() < 0.5)
        param->type = FrameCuboid;
    else
        param->type = FrameTetrahedron;

    for (int iAxis = 3;
        iAxis--;) {

        param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

        for (int iComp = 3;
            iComp--;) {

            param->comp[iComp][iAxis] =
                -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

        }

    }

    param = &paramQ;

}

// Calculate the determinant of the Frames' components matrix
double detP =
    paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2] -
    paramP.comp[1][2] * paramP.comp[2][1]) -
    paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2] -
    paramP.comp[0][2] * paramP.comp[2][1]) +
    paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2] -
    paramP.comp[0][2] * paramP.comp[1][1]);

double detQ =
    paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2] -
    paramQ.comp[1][2] * paramQ.comp[2][1]) -
    paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2] -
    paramQ.comp[0][2] * paramQ.comp[2][1]) +
    paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2] -
    paramQ.comp[0][2] * paramQ.comp[1][1]);

// If the determinants are not null, ie the Frame are not degenerate
if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Validation3D(
        paramP,
        paramQ);

}

}

// If we reached it means the validation was successfull
// Print results
printf("Validation3D has succeed.\n");
printf("Tested %lu intersections ", nbInter);
printf("and %lu no intersections\n", nbNoInter);

```

```

}

int main(int argc, char** argv) {

    printf("==== 2D =====\n");
    Validate2D();
    printf("\n");
    printf("==== 2D =====\n");
    Validate3D();

    return 0;
}

```

## 6.2 Results

### 6.2.1 2D

### 6.2.2 3D

## 7 Qualification

In this section I introduce the code I've used to qualify the algorithm and its implementation. The qualification consists of running the FMB algorithm on randomly generated pairs of Frame, and check its execution time against the one of running the SAT algorithm on the same pair of Frames.

### 7.1 Code

```

// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>

// Include FMB and SAT algorithm library
#include "fmb2d.h"
#include "fmb3d.h"
#include "sat.h"

// Epsilon for numerical precision
#define EPSILON 0.0001
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 1
// Nb of tests per run
#define NB_TESTS 1000000
// Nb of times the test is run on one pair of frame, used to
// slow down the processus and be able to measure time
#define NB_REPEAT_3D 800
#define NB_REPEAT_2D 1500

```



```

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Helper structure to pass arguments to the Qualification function
typedef struct {
    FrameType type;
    double orig[2];
    double comp[2][2];
} Param2D;
typedef struct {
    FrameType type;
    double orig[3];
    double comp[3][3];
} Param3D;

// Global variables to count nb of tests resulting in intersection
// and no intersection, and min/max/total time of execution for each
double minInter;
double maxInter;
double sumInter;
unsigned long countInter;
double minNoInter;
double maxNoInter;
double sumNoInter;
unsigned long countNoInter;

// Qualification function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and measure the time of execution of each
void Qualification2D(
    const Param2D paramP,
    const Param2D paramQ) {

    // Create the two Frames
    Frame2D P =
        Frame2DCreateStatic(
            paramP.type,
            paramP.orig,
            paramP.comp);

    Frame2D Q =
        Frame2DCreateStatic(
            paramQ.type,
            paramQ.orig,
            paramQ.comp);

    // Helper variables to loop on the pair (that, tho) and (tho, that)
    Frame2D* that = &P;
    Frame2D* tho = &Q;

    // Loop on pairs of Frames
    for (int iPair = 2;
        iPair--;) {

        // Declare an array to memorize the results of the repeated
        // test on the same pair,
        // to prevent optimization from the compiler to remove the for loop
        bool isIntersectingFMB[NB_REPEAT_2D] = {false};

        // Start measuring time
        struct timeval start;

```

```

gettimeofday(&start, NULL);

// Run the FMB intersection test
for (int i = NB_REPEAT_2D;
     i--;) {

    isIntersectingFMB[i] =
        FMBTestIntersection2D(
            that,
            tho,
            NULL);
}

// Stop measuring time
struct timeval stop;
gettimeofday(&stop, NULL);

// Calculate the delay of execution
unsigned long deltausFMB = 0;
if (stop.tv_sec < start.tv_sec) {
    printf("time warps, try again\n");
    exit(0);
}
if (stop.tv_sec > start.tv_sec + 1) {
    printf("deltausFMB >> 1s, decrease NB_REPEAT\n");
    exit(0);
}
if (stop.tv_usec < start.tv_usec) {
    deltausFMB = stop.tv_sec - start.tv_sec;
    deltausFMB += stop.tv_usec + 1000000 - start.tv_usec;
} else {
    deltausFMB = stop.tv_usec - start.tv_usec;
}

// Declare an array to memorize the results of the repeated
// test on the same pair,
// to prevent optimization from the compiler to remove the for loop
bool isIntersectingSAT[NB_REPEAT_2D] = {false};

// Start measuring time
gettimeofday(&start, NULL);

// Run the FMB intersection test
for (int i = NB_REPEAT_2D;
     i--;) {

    isIntersectingSAT[i] =
        SATTestIntersection2D(
            that,
            tho);
}

// Stop measuring time
gettimeofday(&stop, NULL);

// Calculate the delay of execution
unsigned long deltausSAT = 0;
if (stop.tv_sec < start.tv_sec) {
    printf("time warps, try again\n");
    exit(0);
}

```

```

if (stop.tv_sec > start.tv_sec + 1) {
    printf("deltausSAT >> 1s, decrease NB_REPEAT\n");
    exit(0);
}
if (stop.tv_usec < start.tv_usec) {
    deltausSAT = stop.tv_sec - start.tv_sec;
    deltausSAT += stop.tv_usec + 1000000 - start.tv_usec;
} else {
    deltausSAT = stop.tv_usec - start.tv_usec;
}

// If the delays are greater than 10ms
if (deltausFMB >= 10 && deltausSAT >= 10) {

    // If FMB and SAT disagrees
    if (isIntersectingFMB[0] != isIntersectingSAT[0]) {

        printf("Qualification has failed\n");
        Frame2DPrint(that);
        printf(" against ");
        Frame2DPrint(tho);
        printf("\n");
        printf("FMB : ");
        if (isIntersectingFMB == false)
            printf("no ");
        printf("intersection\n");
        printf("SAT : ");
        if (isIntersectingSAT == false)
            printf("no ");
        printf("intersection\n");

        // Stop the qualification test
        exit(0);

    }

    // Get the ratio of execution time
    double ratio = ((double)deltausFMB) / ((double)deltausSAT);

    // If the Frames intersect
    if (isIntersectingSAT[0] == true) {

        // Update the counters
        if (countInter == 0) {

            minInter = ratio;
            maxInter = ratio;

        } else {

            if (minInter > ratio)
                minInter = ratio;
            if (maxInter < ratio)
                maxInter = ratio;

        }
        sumInter += ratio;
        ++countInter;

    }

    // Else, the Frames do not intersect
} else {

```

```

        // Update the counters
        if (countNoInter == 0) {

            minNoInter = ratio;
            maxNoInter = ratio;

        } else {

            if (minNoInter > ratio)
                minNoInter = ratio;
            if (maxNoInter < ratio)
                maxNoInter = ratio;

        }
        sumNoInter += ratio;
        ++countNoInter;

    }

    // Else, if time of execution for FMB was less than a 10ms
    } else if (deltausFMB < 10) {

        printf("deltausFMB < 10ms, increase NB_REPEAT\n");
        exit(0);

    // Else, if time of execution for SAT was less than a 10ms
    } else if (deltausSAT < 10) {

        printf("deltausSAT < 10ms, increase NB_REPEAT\n");
        exit(0);

    }

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

}

}

void Qualification3D(
    const Param3D paramP,
    const Param3D paramQ) {

    // Create the two Frames
    Frame3D P =
        Frame3DCreateStatic(
            paramP.type,
            paramP.orig,
            paramP.comp);

    Frame3D Q =
        Frame3DCreateStatic(
            paramQ.type,
            paramQ.orig,
            paramQ.comp);

    // Helper variables to loop on the pair (that, tho) and (tho, that)
    Frame3D* that = &P;
    Frame3D* tho = &Q;

```

```

// Loop on pairs of Frames
for (int iPair = 2;
    iPair--;) {

    // Declare an array to memorize the results of the repeated
    // test on the same pair,
    // to prevent optimization from the compiler to remove the for loop
    bool isIntersectingFMB[NB_REPEAT_3D] = {false};

    // Start measuring time
    struct timeval start;
    gettimeofday(&start, NULL);

    // Run the FMB intersection test
    for (int i = NB_REPEAT_3D;
        i--;) {

        isIntersectingFMB[i] =
            FMBTestIntersection3D(
                that,
                tho,
                NULL);
    }

    // Stop measuring time
    struct timeval stop;
    gettimeofday(&stop, NULL);

    // Calculate the delay of execution
    unsigned long deltausFMB = 0;
    if (stop.tv_sec < start.tv_sec) {
        printf("time warps, try again\n");
        exit(0);
    }
    if (stop.tv_sec > start.tv_sec + 1) {
        printf("deltausFMB >> 1s, decrease NB_REPEAT\n");
        exit(0);
    }
    if (stop.tv_usec < start.tv_usec) {
        deltausFMB = stop.tv_sec - start.tv_sec;
        deltausFMB += stop.tv_usec + 1000000 - start.tv_usec;
    } else {
        deltausFMB = stop.tv_usec - start.tv_usec;
    }

    // Declare an array to memorize the results of the repeated
    // test on the same pair,
    // to prevent optimization from the compiler to remove the for loop
    bool isIntersectingSAT[NB_REPEAT_3D] = {false};

    // Start measuring time
    gettimeofday(&start, NULL);

    // Run the FMB intersection test
    for (int i = NB_REPEAT_3D;
        i--;) {

        isIntersectingSAT[i] =
            SATTestIntersection3D(
                that,
                tho);
    }
}

```

```

}

// Stop measuring time
gettimeofday(&stop, NULL);

// Calculate the delay of execution
unsigned long deltausSAT = 0;
if (stop.tv_sec < start.tv_sec) {
    printf("time warps, try again\n");
    exit(0);
}
if (stop.tv_sec > start.tv_sec + 1) {
    printf("deltausSAT >> 1s, decrease NB_REPEAT\n");
    exit(0);
}
if (stop.tv_usec < start.tv_usec) {
    deltausSAT = stop.tv_sec - start.tv_sec;
    deltausSAT += stop.tv_usec + 1000000 - start.tv_usec;
} else {
    deltausSAT = stop.tv_usec - start.tv_usec;
}

// If the delays are greater than 10ms
if (deltausFMB >= 10 && deltausSAT >= 10) {

    // If FMB and SAT disagrees
    if (isIntersectingFMB[0] != isIntersectingSAT[0]) {

        printf("Qualification has failed\n");
        Frame3DPrint(that);
        printf(" against ");
        Frame3DPrint(tho);
        printf("\n");
        printf("FMB : ");
        if (isIntersectingFMB == false)
            printf("no ");
        printf("intersection\n");
        printf("SAT : ");
        if (isIntersectingSAT == false)
            printf("no ");
        printf("intersection\n");

        // Stop the qualification test
        exit(0);
    }

    // Get the ratio of execution time
    double ratio = ((double)deltausFMB) / ((double)deltausSAT);

    // If the Frames intersect
    if (isIntersectingSAT[0] == true) {

        // Update the counters
        if (countInter == 0) {

            minInter = ratio;
            maxInter = ratio;

        } else {

            if (minInter > ratio)

```

```

        minInter = ratio;
        if (maxInter < ratio)
            maxInter = ratio;

    }
    sumInter += ratio;
    ++countInter;

// Else, the Frames do not intersect
} else {

    // Update the counters
    if (countNoInter == 0) {

        minNoInter = ratio;
        maxNoInter = ratio;

    } else {

        if (minNoInter > ratio)
            minNoInter = ratio;
        if (maxNoInter < ratio)
            maxNoInter = ratio;

    }
    sumNoInter += ratio;
    ++countNoInter;

}

// Else, if time of execution for FMB was less than a 10ms
} else if (deltausFMB < 10) {

    printf("deltausFMB < 10ms, increase NB_REPEAT\n");
    exit(0);

// Else, if time of execution for SAT was less than a 10ms
} else if (deltausSAT < 10) {

    printf("deltausSAT < 10ms, increase NB_REPEAT\n");
    exit(0);

}

// Flip the pair of Frames
that = &Q;
tho = &P;

}

}

void Qualify2D(void) {

    // Initialise the random generator
    srandom(time(NULL));

    // Loop on runs
    for (int iRun = 0;
        iRun < NB_RUNS;
        ++iRun) {

```

```

// Initialize counters
minInter = 0.0;
maxInter = 0.0;
sumInter = 0.0;
countInter = 0;
minNoInter = 0.0;
maxNoInter = 0.0;
sumNoInter = 0.0;
countNoInter = 0;

// Declare two variables to memoize the arguments to the
// Qualification function
Param2D paramP;
Param2D paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

    // Create two random Frame definitions
    Param2D* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

        // 50% chance of being a Cuboid or a Tetrahedron
        if (rnd() < 0.5)
            param->type = FrameCuboid;
        else
            param->type = FrameTetrahedron;

        for (int iAxis = 2;
             iAxis--;) {

            param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            for (int iComp = 2;
                 iComp--;) {

                param->comp[iComp][iAxis] =
                    -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            }

        }

        param = &paramQ;
    }

    // Calculate the determinant of the Frames' components matrix

    double detP =
        paramP.comp[0][0] * paramP.comp[1][1] -
        paramP.comp[1][0] * paramP.comp[0][1];

    double detQ =
        paramQ.comp[0][0] * paramQ.comp[1][1] -
        paramQ.comp[1][0] * paramQ.comp[0][1];

    // If the determinants are not null, ie the Frame are not degenerate
    if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

```



```

        // Run the validation on the two Frames
        Qualification2D(
            paramP,
            paramQ);
    }

}

// Display the results
if (iRun == 0) {

    printf("ratio (timeFMB / timeSAT)\n");
    printf("run\tcountInter\tcountNoInter\t");
    printf("minInter\tavgInter\tmaxInter\t");
    printf("minNoInter\tavgNoInter\tmaxNoInter\t");
    printf("minTotal\tavgTotal\tmaxTotal\n");

}

printf("%d\t%lu\t%lu\t", iRun, countInter, countNoInter);

double avgInter = sumInter / (double)countInter;
printf("%f\t%f\t%f\t", minInter, avgInter, maxInter);

double avgNoInter = sumNoInter / (double)countNoInter;
printf("%f\t%f\t%f\t", minNoInter, avgNoInter, maxNoInter);

double avg =
    (sumInter + sumNoInter) / (double)(countInter + countNoInter);
printf("%f\t%f\t%f\n",
    (minNoInter < minInter ? minNoInter : minInter),
    avg,
    (maxNoInter > maxInter ? maxNoInter : maxInter));
}

}

void Qualify3D(void) {

    // Initialise the random generator
    srand(time(NULL));

    // Loop on runs
    for (int iRun = 0;
        iRun < NB_RUNS;
        ++iRun) {

        // Initialize counters
        minInter = 0.0;
        maxInter = 0.0;
        sumInter = 0.0;
        countInter = 0;
        minNoInter = 0.0;
        maxNoInter = 0.0;
        sumNoInter = 0.0;
        countNoInter = 0;

        // Declare two variables to memoize the arguments to the
        // Qualification function
        Param3D paramP;

```

```

Param3D paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

    // Create two random Frame definitions
    Param3D* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

        // 50% chance of being a Cuboid or a Tetrahedron
        if (rnd() < 0.5)
            param->type = FrameCuboid;
        else
            param->type = FrameTetrahedron;

        for (int iAxis = 3;
             iAxis--;) {

            param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            for (int iComp = 3;
                 iComp--;) {

                param->comp[iComp][iAxis] =
                    -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            }

        }

        param = &paramQ;
    }

    // Calculate the determinant of the Frames' components matrix
    double detP =
        paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2] -
                             paramP.comp[1][2] * paramP.comp[2][1]) -
        paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2] -
                             paramP.comp[0][2] * paramP.comp[2][1]) +
        paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2] -
                             paramP.comp[0][2] * paramP.comp[1][1]);

    double detQ =
        paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2] -
                             paramQ.comp[1][2] * paramQ.comp[2][1]) -
        paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2] -
                             paramQ.comp[0][2] * paramQ.comp[2][1]) +
        paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2] -
                             paramQ.comp[0][2] * paramQ.comp[1][1]);

    // If the determinants are not null, ie the Frame are not degenerate
    if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

        // Run the validation on the two Frames
        Qualification3D(
            paramP,
            paramQ);
    }
}

```

```

    }

    // Display the results
    if (iRun == 0) {

        printf("ratio (timeFMB / timeSAT)\n");
        printf("run\tcountInter\tcountNoInter\t");
        printf("minInter\tavgInter\tmaxInter\t");
        printf("minNoInter\tavgNoInter\tmaxNoInter\t");
        printf("minTotal\tavgTotal\tmaxTotal\n");

    }

    printf("%d\t%lu\t%lu\t", iRun, countInter, countNoInter);

    double avgInter = sumInter / (double)countInter;
    printf("%f\t%f\t%f\t", minInter, avgInter, maxInter);

    double avgNoInter = sumNoInter / (double)countNoInter;
    printf("%f\t%f\t%f\t", minNoInter, avgNoInter, maxNoInter);

    double avg =
        (sumInter + sumNoInter) / (double)(countInter + countNoInter);
    printf("%f\t%f\t%f\n",
        (minNoInter < minInter ? minNoInter : minInter),
        avg,
        (maxNoInter > maxInter ? maxNoInter : maxInter));

    }

}

int main(int argc, char** argv) {

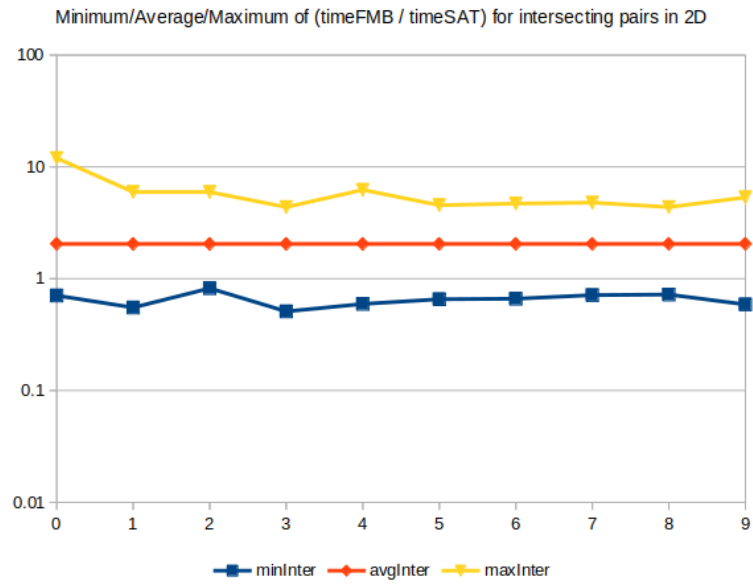
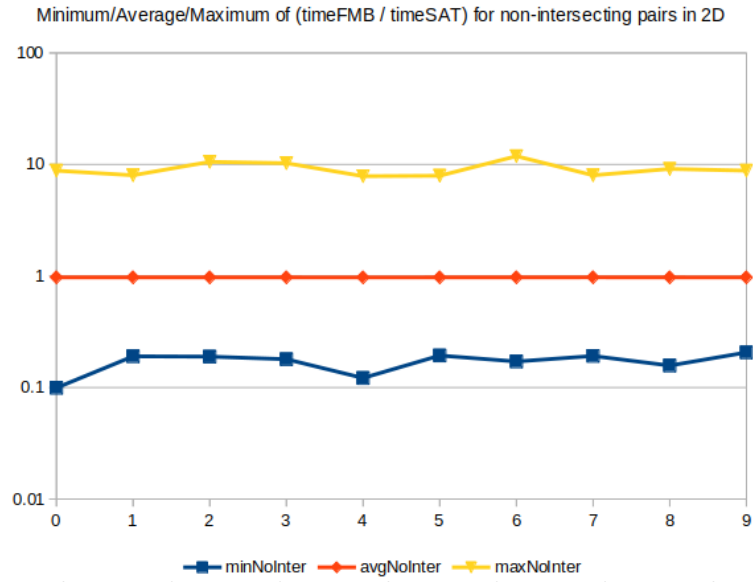
    printf("==== 2D =====\n");
    Qualify2D();
    printf("\n");
    printf("==== 3D =====\n");
    Qualify3D();

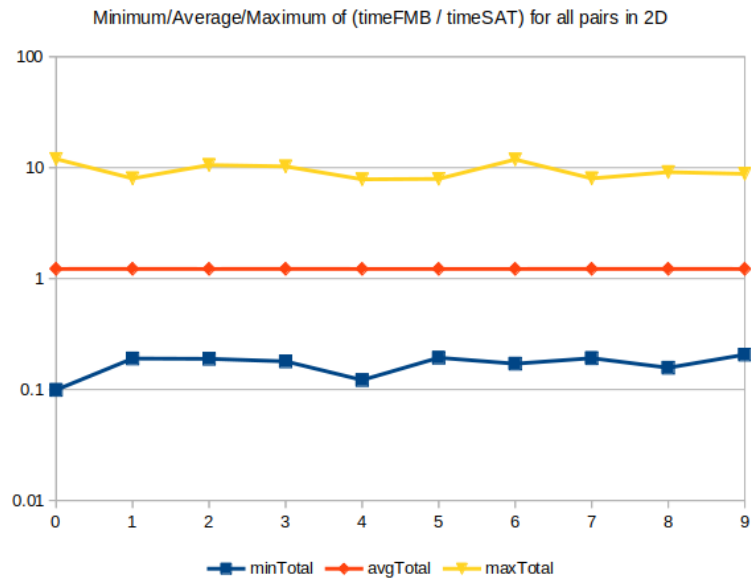
    return 0;
}

```

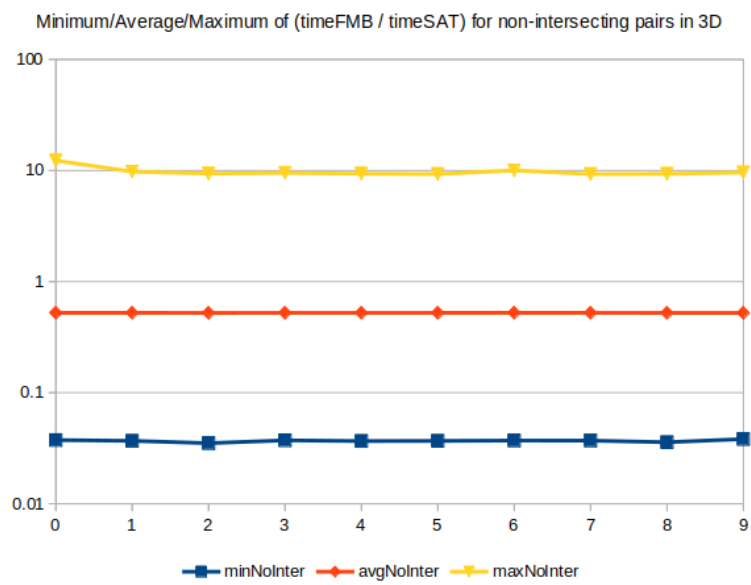
## 7.2 Results

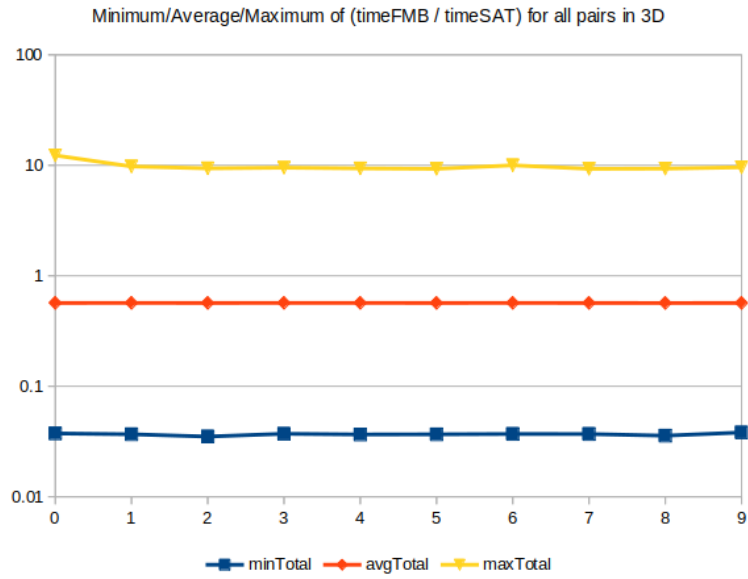
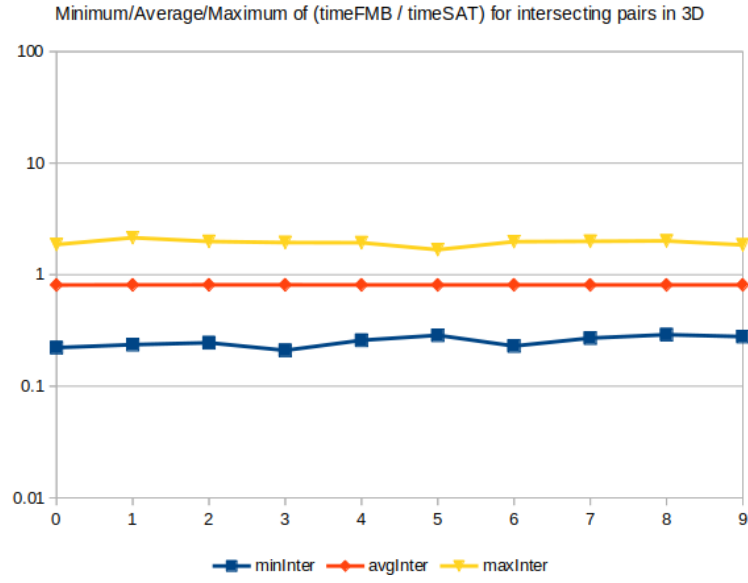
### 7.2.1 2D





### 7.2.2 3D





## 8 Conclusion

The validation proves that the FMB algorithm correctly identifies intersection of pairs of Frames in accordance with the results of the SAT algorithm.

The qualification proves that the FMB algorithm is in average 50% slower than the SAT algorithm in 2D, and 17% faster in 3D.

## 9 Annex

### 9.1 SAT implementation

In this section I introduce the code of the implementation of the SAT algorithm, used to validate and qualify the FMB algorithm.

#### 9.1.1 Header

```
#ifndef __SAT_H_
#define __SAT_H_

#include <stdbool.h>
#include <string.h>
#include "frame.h"

// ----- Functions declaration -----

// Test for intersection between 2D Frame 'that' and 2D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2D(
    const Frame2D* const that,
    const Frame2D* const tho);

// Test for intersection between 3D Frame 'that' and 3D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3D(
    const Frame3D* const that,
    const Frame3D* const tho);

#endif
```

#### 9.1.2 Body

```
#include "sat.h"

// ----- Macros -----

#define EPSILON 0.0000001

// ----- Functions declaration -----

// Check the intersection constraint along one axis
bool CheckAxis3D(
    const Frame3D* const that,
    const Frame3D* const tho,
    const double* const axis);
```

```

// ----- Functions implementation -----

// Test for intersection between 2D Frame 'that' and 2D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2D(
    const Frame2D* const that,
    const Frame2D* const tho) {

    // Declare a variable to loop on Frames and commonalize code
    const Frame2D* frameEdge = that;

    // Loop to commonalize code when checking SAT based on that's edges
    // and then tho's edges
    for (int iTest = 2;
        iTest--;) {

        // Shortcuts
        FrameType frameEdgeType = frameEdge->type;
        const double* frameEdgeCompA = frameEdge->comp[0];
        const double* frameEdgeCompB = frameEdge->comp[1];

        // Declare a variable to memorize the number of edges, by default 2
        int nbEdges = 2;

        // Declare a variable to memorize the third edge in case of
        // tetrahedron
        double thirdEdge[2];

        // If the frame is a tetrahedron
        if (frameEdgeType == FrameTetrahedron) {

            // Initialise the third edge
            thirdEdge[0] = frameEdgeCompB[0] - frameEdgeCompA[0];
            thirdEdge[1] = frameEdgeCompB[1] - frameEdgeCompA[1];

            // Correct the number of edges
            nbEdges = 3;

        }

        // Loop on the frame's edges
        for (int iEdge = nbEdges;
            iEdge--;) {

            // Get the current edge
            const double* edge =
                (iEdge == 2 ? thirdEdge : frameEdge->comp[iEdge]);

            // Declare variables to memorize the boundaries of projection
            // of the two frames on the current edge
            double bdgBoxA[2];
            double bdgBoxB[2];

            // Declare two variables to loop on Frames and commonalize code
            const Frame2D* frame = that;
            double* bdgBox = bdgBoxA;

            // Loop on Frames
            for (int iFrame = 2;
                iFrame--;) {

```



```

// Shortcuts
const double* frameOrig = frame->orig;
const double* frameCompA = frame->comp[0];
const double* frameCompB = frame->comp[1];
FrameType frameType = frame->type;

// Get the number of vertices of frame
int nbVertices = (frameType == FrameTetrahedron ? 3 : 4);

// Declare a variable to memorize if the current vertex is
// the first in the loop, used to initialize the boundaries
bool firstVertex = true;

// Loop on vertices of the frame
for (int iVertex = nbVertices;
     iVertex--;) {

    // Get the vertex
    double vertex[2];
    vertex[0] = frameOrig[0];
    vertex[1] = frameOrig[1];
    switch (iVertex) {
        case 3:
            vertex[0] += frameCompA[0] + frameCompB[0];
            vertex[1] += frameCompA[1] + frameCompB[1];
            break;
        case 2:
            vertex[0] += frameCompA[0];
            vertex[1] += frameCompA[1];
            break;
        case 1:
            vertex[0] += frameCompB[0];
            vertex[1] += frameCompB[1];
            break;
        default:
            break;
    }

    // Get the projection of the vertex on the normal of the edge
    // Orientation of the normal doesn't matter, so we
    // use arbitrarily the normal (edge[1], -edge[0])
    double proj = vertex[0] * edge[1] - vertex[1] * edge[0];

    // If it's the first vertex
    if (firstVertex == true) {

        // Initialize the boundaries of the projection of the
        // Frame on the edge
        bdgBox[0] = proj;
        bdgBox[1] = proj;

        // Update the flag to memorize we did the first vertex
        firstVertex = false;
    }

    // Else, it's not the first vertex
    } else {

        // Update the boundaries of the projection of the Frame on
        // the edge
        if (bdgBox[0] > proj)
            bdgBox[0] = proj;
    }
}

```

```

        if (bdgBox[1] < proj)
            bdgBox[1] = proj;

    }

}

// Switch the frame to check the vertices of the second Frame
frame = tho;
bdgBox = bdgBoxB;

}

// If the projections of the two frames on the edge are
// not intersecting
if (bdgBoxB[1] < bdgBoxA[0] ||
    bdgBoxA[1] < bdgBoxB[0]) {

    // There exists an axis which separates the Frames,
    // thus they are not in intersection
    return false;

}

}

// Switch the frames to test against the second Frame's edges
frameEdge = tho;

}

// If we reaches here, it means the two Frames are intersecting
return true;

}

// Test for intersection between 3D Frame 'that' and 3D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3D(
    const Frame3D* const that,
    const Frame3D* const tho) {

    // Declare two variables to memorize the opposite edges in case
    // of tetrahedron
    double oppEdgesThat[3][3];
    double oppEdgesTho[3][3];

    // Declare two variables to memorize the number of edges, by default 3
    int nbEdgesThat = 3;
    int nbEdgesTho = 3;

    // If the first Frame is a tetrahedron
    if (that->type == FrameTetrahedron) {

        // Shortcuts
        const double* frameCompA = that->comp[0];
        const double* frameCompB = that->comp[1];
        const double* frameCompC = that->comp[2];

        // Initialise the opposite edges
        oppEdgesThat[0][0] = frameCompB[0] - frameCompA[0];
        oppEdgesThat[0][1] = frameCompB[1] - frameCompA[1];

```

```

    oppEdgesThat[0][2] = frameCompB[2] - frameCompA[2];

    oppEdgesThat[1][0] = frameCompB[0] - frameCompC[0];
    oppEdgesThat[1][1] = frameCompB[1] - frameCompC[1];
    oppEdgesThat[1][2] = frameCompB[2] - frameCompC[2];

    oppEdgesThat[2][0] = frameCompC[0] - frameCompA[0];
    oppEdgesThat[2][1] = frameCompC[1] - frameCompA[1];
    oppEdgesThat[2][2] = frameCompC[2] - frameCompA[2];

    // Correct the number of edges
    nbEdgesThat = 6;
}

// If the second Frame is a tetrahedron
if (tho->type == FrameTetrahedron) {

    // Shortcuts
    const double* frameCompA = tho->comp[0];
    const double* frameCompB = tho->comp[1];
    const double* frameCompC = tho->comp[2];

    // Initialise the opposite edges
    oppEdgesTho[0][0] = frameCompB[0] - frameCompA[0];
    oppEdgesTho[0][1] = frameCompB[1] - frameCompA[1];
    oppEdgesTho[0][2] = frameCompB[2] - frameCompA[2];

    oppEdgesTho[1][0] = frameCompB[0] - frameCompC[0];
    oppEdgesTho[1][1] = frameCompB[1] - frameCompC[1];
    oppEdgesTho[1][2] = frameCompB[2] - frameCompC[2];

    oppEdgesTho[2][0] = frameCompC[0] - frameCompA[0];
    oppEdgesTho[2][1] = frameCompC[1] - frameCompA[1];
    oppEdgesTho[2][2] = frameCompC[2] - frameCompA[2];

    // Correct the number of edges
    nbEdgesTho = 6;
}

// Declare variables to loop on Frames and commonalize code
const Frame3D* frame = that;
const double (*oppEdgesA)[3] = oppEdgesThat;

// Loop to commonalize code when checking SAT based on that's edges
// and then tho's edges
for (int iPair = 2;
     iPair--;) {

    // Shortcuts
    FrameType frameType = frame->type;
    const double* frameCompA = frame->comp[0];
    const double* frameCompB = frame->comp[1];
    const double* frameCompC = frame->comp[2];

    // Declare a variable to memorize the number of faces, by default 3
    int nbFaces = 3;

    // Declare a variable to memorize the normal to faces
    // Arrangement is normFaces[iFace][iAxis]
    double normFaces[4][3];

```

```

// Initialise the normal to faces
normFaces[0][0] =
    frameCompA[1] * frameCompB[2] -
    frameCompA[2] * frameCompB[1];
normFaces[0][1] =
    frameCompA[2] * frameCompB[0] -
    frameCompA[0] * frameCompB[2];
normFaces[0][2] =
    frameCompA[0] * frameCompB[1] -
    frameCompA[1] * frameCompB[0];

normFaces[1][0] =
    frameCompA[1] * frameCompC[2] -
    frameCompA[2] * frameCompC[1];
normFaces[1][1] =
    frameCompA[2] * frameCompC[0] -
    frameCompA[0] * frameCompC[2];
normFaces[1][2] =
    frameCompA[0] * frameCompC[1] -
    frameCompA[1] * frameCompC[0];

normFaces[2][0] =
    frameCompC[1] * frameCompB[2] -
    frameCompC[2] * frameCompB[1];
normFaces[2][1] =
    frameCompC[2] * frameCompB[0] -
    frameCompC[0] * frameCompB[2];
normFaces[2][2] =
    frameCompC[0] * frameCompB[1] -
    frameCompC[1] * frameCompB[0];

// If the frame is a tetrahedron
if (frameType == FrameTetrahedron) {

    // Shortcuts
    const double* oppEdgeA = oppEdgesA[0];
    const double* oppEdgeB = oppEdgesA[1];

    // Initialise the normal to the opposite face
    normFaces[3][0] =
        oppEdgeA[1] * oppEdgeB[2] -
        oppEdgeA[2] * oppEdgeB[1];
    normFaces[3][1] =
        oppEdgeA[2] * oppEdgeB[0] -
        oppEdgeA[0] * oppEdgeB[2];
    normFaces[3][2] =
        oppEdgeA[0] * oppEdgeB[1] -
        oppEdgeA[1] * oppEdgeB[0];

    // Correct the number of faces
    nbFaces = 4;

}

// Loop on the frame's faces
for (int iFace = nbFaces;
    iFace--;) {

    // Check against the current face's normal
    bool isIntersection =
        CheckAxis3D(

```

```

        that,
        tho,
        normFaces[iFace]));

// If the axis is separating the Frames
if (isIntersection == false) {

    // The Frames are not in intersection,
    // terminate the test
    return false;

}

}

// Switch the frame to test against the second Frame
frame = tho;
oppEdgesA = oppEdgesTho;

}

// Loop on the pair of edges between the two frames
for (int iEdgeThat = nbEdgesThat;
     iEdgeThat--;) {

    // Get the first edge
    const double* edgeThat =
        (iEdgeThat < 3 ?
         that->comp[iEdgeThat] :
         oppEdgesThat[iEdgeThat - 3]);

    for (int iEdgeTho = nbEdgesTho;
         iEdgeTho--;) {

        // Get the second edge
        const double* edgeTho =
            (iEdgeTho < 3 ?
             tho->comp[iEdgeTho] :
             oppEdgesTho[iEdgeTho - 3]);

        // Get the cross product of the two edges
        double axis[3];
        axis[0] = edgeThat[1] * edgeTho[2] - edgeThat[2] * edgeTho[1];
        axis[1] = edgeThat[2] * edgeTho[0] - edgeThat[0] * edgeTho[2];
        axis[2] = edgeThat[0] * edgeTho[1] - edgeThat[1] * edgeTho[0];

        // Check against the cross product of the two edges
        bool isIntersection =
            CheckAxis3D(
                that,
                tho,
                axis);

        // If the axis is separating the Frames
        if (isIntersection == false) {

            // The Frames are not in intersection,
            // terminate the test
            return false;

        }

    }

}

```

```

    }

}

// If we reaches here, it means the two Frames are intersecting
return true;

}

// Check the intersection constraint for Frames 'that' and 'tho'
// relatively to 'axis'
bool CheckAxis3D(
    const Frame3D* const that,
    const Frame3D* const tho,
    const double* const axis) {

    // Declare variables to memorize the boundaries of projection
    // of the two frames on the current edge
    double bdgBoxA[2];
    double bdgBoxB[2];

    // Declare two variables to loop on Frames and commonalize code
    const Frame3D* frame = that;
    double* bdgBox = bdgBoxA;

    // Loop on Frames
    for (int iFrame = 2;
        iFrame--;) {

        // Shortcuts
        const double* frameOrig = frame->orig;
        const double* frameCompA = frame->comp[0];
        const double* frameCompB = frame->comp[1];
        const double* frameCompC = frame->comp[2];
        FrameType frameType = frame->type;

        // Get the number of vertices of frame
        int nbVertices = (frameType == FrameTetrahedron ? 4 : 8);

        // Declare a variable to memorize if the current vertex is
        // the first in the loop, used to initialize the boundaries
        bool firstVertex = true;

        // Loop on vertices of the frame
        for (int iVertex = nbVertices;
            iVertex--;) {

            // Get the vertex
            double vertex[3];
            vertex[0] = frameOrig[0];
            vertex[1] = frameOrig[1];
            vertex[2] = frameOrig[2];
            switch (iVertex) {
                case 7:
                    vertex[0] +=
                        frameCompA[0] + frameCompB[0] + frameCompC[0];
                    vertex[1] +=
                        frameCompA[1] + frameCompB[1] + frameCompC[1];
                    vertex[2] +=
                        frameCompA[2] + frameCompB[2] + frameCompC[2];
                    break;
                case 6:

```

```

        vertex[0] += frameCompB[0] + frameCompC[0];
        vertex[1] += frameCompB[1] + frameCompC[1];
        vertex[2] += frameCompB[2] + frameCompC[2];
        break;
    case 5:
        vertex[0] += frameCompA[0] + frameCompC[0];
        vertex[1] += frameCompA[1] + frameCompC[1];
        vertex[2] += frameCompA[2] + frameCompC[2];
        break;
    case 4:
        vertex[0] += frameCompA[0] + frameCompB[0];
        vertex[1] += frameCompA[1] + frameCompB[1];
        vertex[2] += frameCompA[2] + frameCompB[2];
        break;
    case 3:
        vertex[0] += frameCompC[0];
        vertex[1] += frameCompC[1];
        vertex[2] += frameCompC[2];
        break;
    case 2:
        vertex[0] += frameCompB[0];
        vertex[1] += frameCompB[1];
        vertex[2] += frameCompB[2];
        break;
    case 1:
        vertex[0] += frameCompA[0];
        vertex[1] += frameCompA[1];
        vertex[2] += frameCompA[2];
        break;
    default:
        break;
}

// Get the projection of the vertex on the axis
double proj =
    vertex[0] * axis[0] +
    vertex[1] * axis[1] +
    vertex[2] * axis[2];

// If it's the first vertex
if (firstVertex == true) {

    // Initialize the boundaries of the projection of the
    // Frame on the edge
    bdgBox[0] = proj;
    bdgBox[1] = proj;

    // Update the flag to memorize we did the first vertex
    firstVertex = false;

// Else, it's not the first vertex
} else {

    // Update the boundaries of the projection of the Frame on
    // the edge
    if (bdgBox[0] > proj)
        bdgBox[0] = proj;

    if (bdgBox[1] < proj)
        bdgBox[1] = proj;
}
}

```

```

    }

    // Switch the frame to check the vertices of the second Frame
    frame = tho;
    bdgBox = bdgBoxB;

}

// If the projections of the two frames on the edge are
// not intersecting
if (bdgBoxB[1] < bdgBoxA[0] ||
    bdgBoxA[1] < bdgBoxB[0]) {

    // There exists an axis which separates the Frames,
    // thus they are not in intersection
    return false;

}

// If we reaches here the two Frames are in intersection
return true;

}

```

## 9.2 Makefile

In this section I introduce the Makefile used to compile the code given in the previous sections.

```

all : main unitTests validation qualification

BUILD_ARG=-O3
BUILD_ARG_=-ggdb

main : main.o fmb2d.o fmb3d.o fmb2dt.o fmb3dt.o sat.o frame.o Makefile
gcc -o main main.o fmb2d.o fmb3d.o sat.o frame.o

main.o : main.c fmb2d.h fmb3d.h fmb2dt.h fmb3dt.h frame.h Makefile
gcc -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb2d.o fmb3d.o fmb2dt.o fmb3dt.o frame.o Makefile
gcc -o unitTests unitTests.o fmb2d.o fmb3d.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb2d.h fmb3d.h fmb2dt.h fmb3dt.h frame.h Makefile
gcc -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb2d.o fmb3d.o fmb2dt.o fmb3dt.o sat.o frame.o Makefile
gcc -o validation validation.o fmb2d.o fmb3d.o sat.o frame.o $(LINK_ARG)

validation.o : validation.c fmb2d.h fmb3d.h fmb2dt.h fmb3dt.h sat.h frame.h Makefile
gcc -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb2d.o fmb3d.o fmb2dt.o fmb3dt.o sat.o frame.o Makefile
gcc -o qualification qualification.o fmb2d.o fmb3d.o sat.o frame.o $(LINK_ARG)

```



```

qualification.o : qualification.c fmb2d.h fmb3d.h fmb2dt.h fmb3dt.h sat.h frame.h Makefile
gcc -c qualification.c $(BUILD_ARG)

fmb2d.o : fmb2d.c fmb2d.h frame.h Makefile
gcc -c fmb2d.c $(BUILD_ARG)

fmb3d.o : fmb3d.c fmb3d.h frame.h Makefile
gcc -c fmb3d.c $(BUILD_ARG)

fmb2dt.o : fmb2dt.c fmb2dt.h frame.h Makefile
gcc -c fmb2dt.c $(BUILD_ARG)

fmb3dt.o : fmb3dt.c fmb3dt.h frame.h Makefile
gcc -c fmb3dt.c $(BUILD_ARG)

sat.o : sat.c sat.h frame.h Makefile
gcc -c sat.c $(BUILD_ARG)

frame.o : frame.c frame.h Makefile
gcc -c frame.c $(BUILD_ARG)

clean :
rm -f *.o main unitTests validation qualification

valgrind :
valgrind -v --track-origins=yes --leak-check=full \
--gen-suppressions=yes --show-leak-kinds=all ./main

```

## References

- [1] J.J.-B. Fourier. Oeuvres II. Paris, 1890
- [2] T.S. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen*. Thesis, 1936. Reprinted in: *Theodore S. Motzkin: selected papers* (D.Cantor et al., eds.), Birkhäuser, Boston, 1983.
- [3] Dynamic Collision Detection using Oriented Bounding Boxes, David Eberly, Geometric Tools, Redmond WA 98052