# The FMB Algorithm

P. Baillehache

January 12, 2020

**Abstract**

This paper introduces how to perform intersection detection of pair of static/dynamic cuboid/tetrahedron in 2D/3D by using the Fourier-Motzkin elimination method.

# Contents

# Introduction

This paper introduces the FMB (Fourier-Motzkin-Baillehache) algorithm which can be used to perform intersection detection of moving and resting parallelepipeds and triangles in 2D, and cuboids and tetrahedrons in 3D.

The detection result is returned has a boolean (intersection / no intersection), and if there is intersection a bounding box of the intersection.

The two first sections introduce how the problem can be expressed as a system of linear inequation, and its resolution using the Fourier-Motzkin method.

The algorithm of the solution and its implementation in the C programming language are detailed in the four following sections.

The last two sections introduce the validation and qualification in term of relative performance of the FMB algorithm against the SAT algorithm.

# 1 The problem as a system of linear inequations

## 1.1 Notations and definitions

- $[M]_{r,c}$ is the component at column $c$ and row $r$ of the matrix $M$
- $[V]_r$ is the $r$-th component of the vector $\overrightarrow{V}$
- the term "frame" is used indifferently for parallelepiped, triangle, cuboid and tetrahedron.

## 1.2 Static case

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:

$$\mathbb{A} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{A}}.\overrightarrow{X} \end{array} \right\} \tag{1}$$

$$\mathbb{B} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ \overrightarrow{O_{\mathbb{B}}} + C_{\mathbb{B}}.\overrightarrow{X} \end{array} \right\} \tag{2}$$

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

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

$$\mathbb{A} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{A}}.\overrightarrow{X} \end{array} \right\} \tag{3}$$

$$\mathbb{B} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ \overrightarrow{O_{\mathbb{B}}} + C_{\mathbb{B}}.\overrightarrow{X} \end{array} \right\} \tag{4}$$

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:

$$\mathbb{B}_{\mathbb{A}} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ C_{\mathbb{A}}^{-1}.(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}) \end{array} \right\} \tag{5}$$

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

$$\mathbb{B}_{\mathbb{A}} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1}.(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}) \end{array} \right\} \tag{6}$$

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

$$\mathbb{A}_{\mathbb{A}} = \left\{ \overrightarrow{X} \in [0.0, 1.0]^D \right\} \tag{7}$$

and for a tetrahedron:

$$\mathbb{A}_{\mathbb{A}} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \end{array} \right\} \tag{8}$$

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:

$$\left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}\right) \cap [0.0, 1.0]^D \end{array} \right\} \tag{9}$$

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

$$\left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}\right) \cap [0.0, 1.0]^D \end{array} \right\} \tag{10}$$

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

$$\left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}\right) \cap [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} \left[C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}\right)\right]_i \leq 1.0 \end{array} \right\} \tag{11}$$

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

$$\left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1}.(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}) \cap [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} \left[C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}\right)\right]_i \leq 1.0 \end{array} \right\} \tag{12}$$

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

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

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

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

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

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

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

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

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

### 1.3 Dynamic case

If the frames $\mathbb{A}$ and $\mathbb{B}$ are moving linearly along the vectors $\overrightarrow{V_\mathbb{A}}$ and $\overrightarrow{V_\mathbb{B}}$ 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:

$$\mathbb{A} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{A}}.\overrightarrow{X} + \overrightarrow{V_{\mathbb{A}}}.t \end{array} \right\} \tag{17}$$

$$\mathbb{B} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \overrightarrow{O_{\mathbb{B}}} + C_{\mathbb{B}}.\overrightarrow{X} + \overrightarrow{V_{\mathbb{B}}}.t \end{array} \right\} \tag{18}$$

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

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

$$\mathbb{A} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{A}}.\overrightarrow{X} + \overrightarrow{V_{\mathbb{A}}}.t \end{array} \right\} \tag{19}$$

$$\mathbb{B} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ \overrightarrow{O_{\mathbb{B}}} + C_{\mathbb{B}}.\overrightarrow{X} + \overrightarrow{V_{\mathbb{B}}}.t \end{array} \right\} \tag{20}$$

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

$$\mathbb{B}_{\mathbb{A}} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X} + \left(\overrightarrow{V_{\mathbb{B}}} - \overrightarrow{V_{\mathbb{A}}}\right).t\right) \end{array} \right\} \tag{21}$$

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

$$\mathbb{B}_{\mathbb{A}} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X} + \left(\overrightarrow{V_{\mathbb{B}}} - \overrightarrow{V_{\mathbb{A}}}\right).t\right) \end{array} \right\} \tag{22}$$

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

$$\mathbb{A}_{\mathbb{A}} = \left\{ \overrightarrow{X} \in [0.0, 1.0]^D \right\} \tag{23}$$

and for a tetrahedron:

$$\mathbb{A}_{\mathbb{A}} = \left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \end{array} \right\} \tag{24}$$

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:

$$\left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X} + \left(\overrightarrow{V_{\mathbb{B}}} - \overrightarrow{V_{\mathbb{A}}}\right).t\right) \cap [0.0, 1.0]^D \end{array} \right\} \tag{25}$$

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

$$\left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X} + \left(\overrightarrow{V_{\mathbb{B}}} - \overrightarrow{V_{\mathbb{A}}}\right).t\right) \cap [0.0, 1.0]^D \end{array} \right\} \tag{26}$$

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

$$\left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X} + \left(\overrightarrow{V_{\mathbb{B}}} - \overrightarrow{V_{\mathbb{A}}}\right).t\right) \cap [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} \left[C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}\right)\right]_i \leq 1.0 \end{array} \right\} \tag{27}$$

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

$$\left\{ \begin{array}{c} \overrightarrow{X} \in [0.0, 1.0]^D \\ t \in [0.0, 1.0] \\ \sum_{i=0}^{D-1} [X]_i \leq 1.0 \\ C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X} + \left(\overrightarrow{V_{\mathbb{B}}} - \overrightarrow{V_{\mathbb{A}}}\right).t\right) \cap [0.0, 1.0]^D \\ \sum_{i=0}^{D-1} \left[C_{\mathbb{A}}^{-1}.\left(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}} + C_{\mathbb{B}}.\overrightarrow{X}\right)\right]_i \leq 1.0 \end{array} \right\} \tag{28}$$

These lead to the following systems of linear inequations, given the three shortcuts $\overrightarrow{O_{\mathbb{B}_{\mathbb{A}}}} = C_{\mathbb{A}}^{-1}.(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}})$, $\overrightarrow{V_{\mathbb{B}_{\mathbb{A}}}} = C_{\mathbb{A}}^{-1}.(\overrightarrow{V_{\mathbb{B}}} - \overrightarrow{V_{\mathbb{A}}})$ and $C_{\mathbb{B}_{\mathbb{A}}} = C_{\mathbb{A}}^{-1}.C_{\mathbb{B}}$.

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

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

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

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

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

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

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

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

# 2 Resolution of the problem by Fourier-Motzkin method

## 2.1 The 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

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

with

$$\begin{array}{c} i \in 1, 2, ..., m \\ j \in 1, 2, ..., n \\ x_i \in \mathbb{R} \\ a_{ij} \in \mathbb{R} \\ b_j \in \mathbb{R} \end{array} \tag{34}$$

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

$$\left\{\begin{array}{cllllc} x_1 & +a'_{i2}.x_2 & +\cdots & +a'_{in}.x_n & \leq b'_i & (i \in \mathcal{I}_+) \\ & a_{i2}.x_2 & +\cdots & +a_{in}.x_n & \leq b_i & (i \in \mathcal{I}_0) \\ -x_1 & +a'_{i2}.x_2 & +\cdots & +a'_{in}.x_n & \leq b'_i & (i \in \mathcal{I}_-) \end{array}\right. \tag{35}$$

where

$$\begin{array}{c} \mathcal{I}_+ = \{i : a_{i1} > 0.0\} \\ \mathcal{I}_0 = \{i : a_{i1} = 0.0\} \\ \mathcal{I}_- = \{i : a_{i1} < 0.0\} \\ a'_{ij} = a_{ij}/|a_{i1}| \\ b'_i = b_i/|a_{i1}| \end{array}$$

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

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

and

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

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

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

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 $x_1$.

## 2.2 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 $\overrightarrow{S}$ 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 $\overrightarrow{S'}$ in the real world's coordinates system as follow:

$$\overrightarrow{S'} = \overrightarrow{O_{\mathbb{B}}} + C_{\mathbb{B}}.\overrightarrow{S} \tag{39}$$

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

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

$$Y < \sum_{i \in I^-} a_i \tag{40}$$

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

## 2.3 About the size of system of linear inequation

During implementation in languages where the developper needs to manage memory itself the size of the systems (35) resulting from variable elimination is necessary but cannot be forecasted. Instead, a maximum size can be calculated as follow.

Lets call $n_-$, $n_+$ and $n_0$ the size of, respectively, $\mathcal{I}_-$, $\mathcal{I}_+$ and $\mathcal{I}_0$, and $N$ the number of inequalities in the original system and $N'$ the number inequalities in the resulting system. We have:

$$n_- + n_+ + n_0 = N \tag{41}$$

and

$$n_-.n_+ + n_0 = N' \tag{42}$$

Now lets define $K = N - n_0$, then we have:

$$n_- + n_+ = K \tag{43}$$

then,

$$n_-.n_+ = n_-(K - n_-) \tag{44}$$

then,

$$n_-.n_+ = K.n_-n_-^2 \tag{45}$$

The right part is polynomial whose maximum is reached for $n_- = K/2$. Then,

$$n_-.n_+ \leq K^2/2 - K^2/4 \tag{46}$$

or,

$$n_-.n_+ \leq K^2/4 \tag{47}$$

and putting back the definition of $K$

$$n_-.n_+ \leq (N - n_0)^2/4 \tag{48}$$

which is also

$$n_-.n_+ \leq N^2/4 \tag{49}$$

From (42) we get,

$$N' \leq N^2/4 - n_0 \tag{50}$$

and getting rid of the $n_0$ knowing that $n_0 \geq 0$,

$$N' \leq N^2/4 \tag{51}$$

The maximum number of inequation in the initial system is defined for each case (2D/3D, static/dynamic) in the previous section. This leads to the following maximum number of inequations:

| | $N$ | $N'$ | $N''$ | $N'''$ |
|---|---|---|---|---|
| $2Dstatic$ | 8 | 16 | | |
| $2Ddynamic$ | 10 | 25 | 157 | |
| $3Dstatic$ | 12 | 36 | 324 | |
| $3Ddynamic$ | 14 | 49 | 601 | 90301 |

# 3 Algorithms of the solution

In this section I introduce the algorithms of the solution of the previous section for each case (static/dynamic and 2D/3D), and the algorithms to manipulate the structure used to represent the cuboid and tetrahedron.

Algorithms are given in pseudo code, and consequently without any optimization based on properties of one given language. One can refer to the C implementation in the following sections for possible optimization in this language.

Algorithms are also given independantly from each other. Code commonalization may be possible if one plans to gather several cases together, but this is dependant of the implementation and thus left to the developper responsibility.

## 3.1 2D static

```
ENUM FrameType
  FrameCuboid,
  FrameTetrahedron
END ENUM

STRUCT AABB2D
  // x,y
  real min[2]
  real max[2]
END STRUCT

STRUCT Frame2D
  FrameType type
  real orig[2]
  // comp[iComp][iAxis]
  real comp[2][2]
  AABB2D bdgBox
  real invComp[2][2]
END STRUCT

FUNCTION powi(base, exp)
    res = 1
    FOR i=0..(exp - 1)
      res = res * base
    END FOR
    RETURN res
END FUNCTION

FUNCTION Frame2DPrint(that)
  IF that.type == FrameTetrahedron
    PRINT "T"
  ELSE IF that.type == FrameCuboid
    PRINT "C"
```

```
  END IF
  PRINT "o("
  FOR i = 0..1
    PRINT that.orig[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  comp = ['x','y']
  FOR j = 0..1
    PRINT ") " comp[j] "("
    FOR i = 0..1
      PRINT that.comp[j][i]
      IF i < 1
        PRINT ","
      END IF
    END FOR
  END FOR
  PRINT ")"
END FUNCTION

FUNCTION AABB2DPrint(that)
  PRINT "minXY("
  FOR i = 0..1
    PRINT that.min[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  PRINT ")-maxXY("
  FOR i = 0..1
    PRINT that.max[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  PRINT ")"
END FUNCTION

FUNCTION Frame2DExportBdgBox(that, bdgBox, bdgBoxProj)
  FOR i = 0..1
    bdgBoxProj.max[i] = that.orig[i]
    FOR j = 0..1
      bdgBoxProj.max[i] =
        bdgBoxProj.max[i] + that.comp[j][i] * bdgBox.min[j]
    END FOR
    bdgBoxProj.min[i] = bdgBoxProj.max[i]
  END FOR
  nbVertices = powi(2, 2)
  FOR iVertex = 1..(nbVertices - 1)
    FOR i = 0..1
      IF (iVertex & (1 << i)) == TRUE
        v[i] = bdgBox.max[i]
      ELSE
        v[i] = bdgBox.min[i]
      END IF
    END FOR
    FOR i = 0..1
      w[i] = that.orig[i]
      FOR j = 0..1
        w[i] = w[i] + that.comp[j][i] * v[j]
      END FOR
```

```
    END FOR
    FOR i = 0..1
      IF bdgBoxProj.min[i] > w[i]
        bdgBoxProj.min[i] = w[i]
      END IF
      IF bdgBoxProj.max[i] < w[i]
        bdgBoxProj.max[i] = w[i]
      END IF
    END FOR
  END FOR
END FUNCTION

FUNCTION Frame2DImportFrame(P, Q, Qp)
  FOR i = 0..1
    v[i] = Q.orig[i] - P.orig[i]
  END FOR
  FOR i = 0..1
    Qp.orig[i] = 0.0
    FOR j = 0..1
      Qp.orig[i] = Qp.orig[i] + P.invComp[j][i] * v[j]
      Qp.comp[j][i] = 0.0
      FOR k = 0..1
        Qp.comp[j][i] = Qp.comp[j][i] + P.invComp[k][i] * Q.comp[j][k]
      END FOR
    END FOR
  END FOR
END FUNCTION

FUNCTION Frame2DUpdateInv(that)
  det = that.comp[0][0] * that.comp[1][1] -
    that.comp[1][0] * that.comp[0][1]
  that.invComp[0][0] = that.comp[1][1] / det
  that.invComp[0][1] = -that.comp[0][1] / det
  that.invComp[1][0] = -that.comp[1][0] / det
  that.invComp[1][1] = that.comp[0][0] / det
END FUNCTION

FUNCTION Frame2DCreateStatic(type, orig, comp)
  that.type = type
  FOR iAxis = 0..1
    that.orig[iAxis] = orig[iAxis]
    FOR iComp = 0..1
      that.comp[iComp][iAxis] = comp[iComp][iAxis]
    END FOR
  END FOR
  FOR iAxis = 0..1
    min = orig[iAxis]
    max = orig[iAxis]
    FOR iComp = 0..1
      IF that.type == FrameCuboid
        IF that.comp[iComp][iAxis] < 0.0
          min += that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0
          max += that.comp[iComp][iAxis]
        END IF
      ELSE IF that.type == FrameTetrahedron
        IF that.comp[iComp][iAxis] < 0.0 AND
          min > orig[iAxis] + that.comp[iComp][iAxis]
          min = orig[iAxis] + that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0 AND
```

```
          max < orig[iAxis] + that.comp[iComp][iAxis]
          max = orig[iAxis] + that.comp[iComp][iAxis]
        END IF
      END IF
    END FOR
    that.bdgBox.min[iAxis] = min
    that.bdgBox.max[iAxis] = max
  END FOR
  Frame2DUpdateInv(that)
  RETURN that
END FUNCTION

FUNCTION Sgn(v)
  IF 0.0 < v
    a = 1
  ELSE
    a = 0
  END IF
  IF v < 0.0
    b = 1
  ELSE
    b = 0
  END IF
  RETURN A - B
END FUNCTION

FUNCTION Neg(x)
  IF x < 0.0
    RETURN x
  ELSE
    RETURN 0.0
  END IF
END FUNCTION

FST_VAR = 0
SND_VAR = 1

FUNCTION ElimVar2D(iVar, M, Y, nbRows, nbCols, Mp, Yp, nbRemainRows)
  nbRemainRows = 0
  FOR iRow = 0..(nbRows - 2)
    IF M[iRow][iVar] <> 0.0
      FOR jRow = (iRow + 1)..(nbRows - 1)
        IF sgn(M[iRow][iVar]) <> sgn(M[jRow][iVar]) AND
            M[jRow][iVar] <> 0.0
          sumNegCoeff = 0.0
          jCol = 0
          FOR iCol = 0..(nbCols - 1)
            IF iCol <> iVar
              Mp[nbRemainRows][jCol] =
                M[iRow][iCol] / fabs(M[iRow][iVar]) +
                M[jRow][iCol] / fabs(M[jRow][iVar])
              sumNegCoeff = sumNegCoeff + neg(Mp[nbRemainRows][jCol])
              jCol = jCol + 1
            END IF
          END FOR
          Yp[nbRemainRows] =
            Y[iRow] / fabs(M[iRow][iVar]) +
            Y[jRow] / fabs(M[jRow][iVar])
          IF Yp[nbRemainRows] < sumNegCoeff
            RETURN TRUE
          END IF
          nbRemainRows = nbRemainRows + 1
```

```
        END IF
      END FOR
    END IF
  END FOR
  FOR iRow = 0..(nbRows - 1)
    IF M[iRow][iVar] == 0.0
      jCol = 0
      FOR iCol = 0..(nbCols - 1)
        IF iCol <> iVar
          Mp[nbRemainRows][jCol] = M[iRow][iCol]
          jCol = jCol + 1
        END IF
      END FOR
      Yp[nbRemainRows] = Y[iRow]
      nbRemainRows = nbRemainRows + 1
    END IF
  END FOR
  RETURN FALSE
END FUNCTION

FUNCTION GetBound2D(iVar, M, Y, nbRows, bdgBox)
  bdgBox.min[iVar] = 0.0
  bdgBox.max[iVar] = 1.0
  FOR jRow = 0..(nbRows - 1)
    IF M[jRow][0] > 0.0
      y = Y[jRow] / M[jRow][0]
      IF bdgBox.max[iVar] > y
        bdgBox.max[iVar] = y
      END IF
    ELSE IF M[jRow][0] < 0.0
      y = Y[jRow] / M[jRow][0]
      IF bdgBox.min[iVar] < y
        bdgBox.min[iVar] = y
      END IF
    END IF
  END FOR
END FUNCTION

FUNCTION FMBTestIntersection2D(that, tho, bdgBox)
  Frame2DImportFrame(that, tho, &thoProj)
  M[0][0] = -thoProj.comp[0][0]
  M[0][1] = -thoProj.comp[1][0]
  Y[0] = thoProj.orig[0]
  IF Y[0] < neg(M[0][0]) + neg(M[0][1])
    RETURN FALSE
  END IF
  M[1][0] = -thoProj.comp[0][1]
  M[1][1] = -thoProj.comp[1][1]
  Y[1] = thoProj.orig[1]
  IF Y[1] < neg(M[1][0]) + neg(M[1][1])
    RETURN FALSE
  END IF
  nbRows = 2
  IF that.type == FrameCuboid
    M[nbRows][0] = thoProj.comp[0][0]
    M[nbRows][1] = thoProj.comp[1][0]
    Y[nbRows] = 1.0 - thoProj.orig[0]
    IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1])
      RETURN FALSE
    END IF
    nbRows = nbRows + 1
    M[nbRows][0] = thoProj.comp[0][1]
```

```
    M[nbRows][1] = thoProj.comp[1][1]
    Y[nbRows] = 1.0 - thoProj.orig[1]
    IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1])
      RETURN FALSE
    END IF
    nbRows = nbRows + 1
  ELSE
    M[nbRows][0] = thoProj.comp[0][0] + thoProj.comp[0][1]
    M[nbRows][1] = thoProj.comp[1][0] + thoProj.comp[1][1]
    Y[nbRows] = 1.0 - thoProj.orig[0] - thoProj.orig[1]
    IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1])
      RETURN FALSE
    END IF
    nbRows = nbRows + 1
  END
  IF tho.type == FrameCuboid
    M[nbRows][0] = 1.0
    M[nbRows][1] = 0.0
    Y[nbRows] = 1.0
    nbRows = nbRows + 1
    M[nbRows][0] = 0.0
    M[nbRows][1] = 1.0
    Y[nbRows] = 1.0
    nbRows = nbRows + 1
  ELSE
    M[nbRows][0] = 1.0
    M[nbRows][1] = 1.0
    Y[nbRows] = 1.0
    nbRows = nbRows + 1
  END
  M[nbRows][0] = -1.0
  M[nbRows][1] = 0.0
  Y[nbRows] = 0.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = -1.0
  Y[nbRows] = 0.0
  nbRows = nbRows + 1
  inconsistency = ElimVar2D(FST_VAR, M, Y, nbRows, 2, Mp, Yp, nbRowsP)
  IF inconsistency == TRUE
    RETURN FALSE
  END
  GetBound2D(SND_VAR, Mp, Yp, nbRowsP, bdgBoxLocal)
  IF bdgBoxLocal.min[SND_VAR] >= bdgBoxLocal.max[SND_VAR]
    RETURN FALSE
  END
  ElimVar2D(SND_VAR, M, Y, nbRows, 2, Mp, Yp, nbRowsP)
  GetBound2D(FST_VAR, Mp, Yp, nbRowsP, bdgBoxLocal)
  bdgBox = bdgBoxLocal
  RETURN TRUE
END

origP2D = [0.0, 0.0]
compP2D = [
  [1.0, 0.0],
  [0.0, 1.0]]
P2D = Frame2DCreateStatic(FrameCuboid, origP2D, compP2D)
origQ2D = [0.0, 0.0]
compQ2D = [
  [1.0, 0.0],
  [0.0, 1.0]]
Q2D = Frame2DCreateStatic(FrameCuboid, origQ2D, compQ2D)
```

```
isIntersecting2D = FMBTestIntersection2D(P2D, Q2D, bdgBox2DLocal)
if isIntersecting2D == TRUE
  PRINT "Intersection detected."
  Frame2DExportBdgBox(Q2D, bdgBox2DLocal, bdgBox2D);
  AABB2DPrint(bdgBox2D)
ELSE
  PRINT "No intersection."
END IF
```

## 3.2 3D static

```
ENUM FrameType
  FrameCuboid,
  FrameTetrahedron
END ENUM

STRUCT AABB3D
  // x,y,z
  real min[3]
  real max[3]
END STRUCT

STRUCT Frame3D
  FrameType type
  real orig[3]
  // comp[iComp][iAxis]
  real comp[3][3]
  AABB3D bdgBox
  real invComp[3][3]
END STRUCT

FUNCTION powi(base, exp)
    res = 1
    FOR i=0..(exp - 1)
      res = res * base
    END FOR
    RETURN res
END FUNCTION

FUNCTION Frame3DPrint(that)
  IF that.type == FrameTetrahedron
    PRINT "T"
  ELSE IF that.type == FrameCuboid
    PRINT "C"
  END IF
  PRINT "o("
  FOR i = 0..2
    PRINT that.orig[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  comp = ['x','y','z']
  FOR j = 0..2
    PRINT ") " comp[j] "("
    FOR i = 0..2
      PRINT that.comp[j][i]
      IF i < 2
        PRINT ","
      END IF
    END FOR
```

```
  END FOR
  PRINT ")"
END FUNCTION

FUNCTION AABB3DPrint(that)
  PRINT "minXYZ("
  FOR i = 0..2
    PRINT that.min[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  PRINT ")-maxXYZ("
  FOR i = 0..2
    PRINT that.max[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  PRINT ")"
END FUNCTION

FUNCTION Frame3DExportBdgBox(that, bdgBox, bdgBoxProj)
  FOR i = 0..2
    bdgBoxProj.max[i] = that.orig[i]
    FOR j = 0..2
      bdgBoxProj.max[i] =
        bdgBoxProj.max[i] + that.comp[j][i] * bdgBox.min[j]
    END FOR
    bdgBoxProj.min[i] = bdgBoxProj.max[i]
  END FOR
  nbVertices = powi(2, 3)
  FOR iVertex = 1..(nbVertices - 1)
    FOR i = 0..2
      IF (iVertex & (1 << i)) == TRUE
        v[i] = bdgBox.max[i]
      ELSE
        v[i] = bdgBox.min[i]
      END IF
    END FOR
    FOR i = 0..2
      w[i] = that.orig[i]
      FOR j = 0..2
        w[i] = w[i] + that.comp[j][i] * v[j]
      END FOR
    END FOR
    FOR i = 0..2
      IF bdgBoxProj.min[i] > w[i]
        bdgBoxProj.min[i] = w[i]
      END IF
      IF bdgBoxProj.max[i] < w[i]
        bdgBoxProj.max[i] = w[i]
      END IF
    END FOR
  END FOR
END FUNCTION

FUNCTION Frame3DImPortFrame(P, Q, Qp)
  FOR i = 0..2
    v[i] = Q.orig[i] - P.orig[i]
  END FOR
  FOR i = 0..2
```

```
    Qp.orig[i] = 0.0
    FOR j = 0..2
      Qp.orig[i] = Qp.orig[i] + P.invComp[j][i] * v[j]
      Qp.comp[j][i] = 0.0
      FOR k = 0..2
        Qp.comp[j][i] = Qp.comp[j][i] + P.invComp[k][i] * Q.comp[j][k]
      END FOR
    END FOR
  END FOR
END FUNCTION

FUNCTION Frame3DUpdateInv(that)
  det =
    that.comp[0][0] * (that.comp[1][1] * that.comp[2][2] -
    that.comp[1][2] * that.comp[2][1]) -
    that.comp[1][0] * (that.comp[0][1] * that.comp[2][2] -
    that.comp[0][2] * that.comp[2][1]) +
    that.comp[2][0] * (that.comp[0][1] * that.comp[1][2] -
    that.comp[0][2] * that.comp[1][1])
  that.invComp[0][0] = (that.comp[1][1] * that.comp[2][2] -
    that.comp[2][1] * that.comp[1][2]) / det
  that.invComp[0][1] = (that.comp[2][1] * that.comp[0][2] -
    that.comp[2][2] * that.comp[0][1]) / det
  that.invComp[0][2] = (that.comp[0][1] * that.comp[1][2] -
    that.comp[0][2] * that.comp[1][1]) / det
  that.invComp[1][0] = (that.comp[2][0] * that.comp[1][2] -
    that.comp[2][2] * that.comp[1][0]) / det
  that.invComp[1][1] = (that.comp[0][0] * that.comp[2][2] -
    that.comp[2][0] * that.comp[0][2]) / det
  that.invComp[1][2] = (that.comp[0][2] * that.comp[1][0] -
    that.comp[1][2] * that.comp[0][0]) / det
  that.invComp[2][0] = (that.comp[1][0] * that.comp[2][1] -
    that.comp[2][0] * that.comp[1][1]) / det
  that.invComp[2][1] = (that.comp[0][1] * that.comp[2][0] -
    that.comp[2][1] * that.comp[0][0]) / det
  that.invComp[2][2] = (that.comp[0][0] * that.comp[1][1] -
    that.comp[1][0] * that.comp[0][1]) / det
END FUNCTION

FUNCTION Frame3DCreateStatic(type, orig, comp)
  that.type = type
  FOR iAxis = 0..2
    that.orig[iAxis] = orig[iAxis]
    FOR iComp = 0..2
      that.comp[iComp][iAxis] = comp[iComp][iAxis]
    END FOR
  END FOR
  FOR iAxis = 0..2
    min = orig[iAxis]
    max = orig[iAxis]
    FOR iComp = 0..2
      IF that.type == FrameCuboid) {
        IF that.comp[iComp][iAxis] < 0.0
          min += that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0
          max += that.comp[iComp][iAxis]
        END IF
      ELSE IF that.type == FrameTetrahedron
        IF that.comp[iComp][iAxis] < 0.0 AND
          min > orig[iAxis] + that.comp[iComp][iAxis]
          min = orig[iAxis] + that.comp[iComp][iAxis]
```

```
        END IF
        IF that.comp[iComp][iAxis] > 0.0 AND
          max < orig[iAxis] + that.comp[iComp][iAxis]
          max = orig[iAxis] + that.comp[iComp][iAxis]
        END IF
      END IF
    END FOR
    that.bdgBox.min[iAxis] = min
    that.bdgBox.max[iAxis] = max
  END FOR
  Frame3DUpdateInv(that)
  RETURN that
END FUNCTION

FUNCTION Sgn(v)
  IF 0.0 < v
    a = 1
  ELSE
    a = 0
  END IF
  IF v < 0.0
    b = 1
  ELSE
    b = 0
  END IF
  RETURN A - B
END FUNCTION

FUNCTION Neg(x)
  IF x < 0.0
    RETURN x
  ELSE
    RETURN 0.0
  END IF
END FUNCTION

FST_VAR = 0
SND_VAR = 1
THD_VAR = 2

FUNCTION ElimVar3D(iVar, M, Y, nbRows, nbCols, Mp, Yp, nbRemainRows)
  nbRemainRows = 0
  FOR iRow = 0..(nbRows - 2)
    IF M[iRow][iVar] <> 0.0
      FOR jRow = (iRow + 1)..(nbRows - 1)
        IF sgn(M[iRow][iVar]) <> sgn(M[jRow][iVar]) AND
            M[jRow][iVar] <> 0.0
          sumNegCoeff = 0.0
          jCol = 0
          FOR iCol = 0..(nbCols - 1)
            IF iCol <> iVar
              Mp[nbRemainRows][jCol] =
                M[iRow][iCol] / fabs(M[iRow][iVar]) +
                M[jRow][iCol] / fabs(M[jRow][iVar])
              sumNegCoeff = sumNegCoeff + neg(Mp[nbRemainRows][jCol])
              jCol = jCol + 1
            END IF
          END FOR
          Yp[nbRemainRows] =
            Y[iRow] / fabs(M[iRow][iVar]) +
            Y[jRow] / fabs(M[jRow][iVar])
          IF Yp[nbRemainRows] < sumNegCoeff
```

```
            RETURN TRUE
          END IF
          nbRemainRows = nbRemainRows + 1
        END IF
      END FOR
    END IF
  END FOR
  FOR iRow = 0..(nbRows - 1)
    IF M[iRow][iVar] == 0.0
      jCol = 0
      FOR iCol = 0..(nbCols - 1)
        IF iCol <> iVar
          Mp[nbRemainRows][jCol] = M[iRow][iCol]
          jCol = jCol + 1
        END IF
      END FOR
      Yp[nbRemainRows] = Y[iRow]
      nbRemainRows = nbRemainRows + 1
    END IF
  END FOR
  RETURN FALSE
END FUNCTION

FUNCTION GetBound3D(iVar, M, Y, nbRows, bdgBox)
  bdgBox.min[iVar] = 0.0
  bdgBox.max[iVar] = 1.0
  FOR jRow = 0..(nbRows - 1)
    IF M[jRow][0] > 0.0
      y = Y[jRow] / M[jRow][0]
      IF bdgBox.max[iVar] > y
        bdgBox.max[iVar] = y
      END IF
    ELSE IF M[jRow][0] < 0.0
      y = Y[jRow] / M[jRow][0]
      IF bdgBox.min[iVar] < y
        bdgBox.min[iVar] = y
      END IF
    END IF
  END FOR
END FUNCTION

FUNCTION FMBTestIntersection3D(that, tho, bdgBox)
  Frame3DImportFrame(that, tho, thoProj)
  M[0][0] = -thoProj.comp[0][0]
  M[0][1] = -thoProj.comp[1][0]
  M[0][2] = -thoProj.comp[2][0]
  Y[0] = thoProj.orig[0]
  IF Y[0] < neg(M[0][0]) + neg(M[0][1]) + neg(M[0][2])
    RETURN FALSE
  END IF
  M[1][0] = -thoProj.comp[0][1]
  M[1][1] = -thoProj.comp[1][1]
  M[1][2] = -thoProj.comp[2][1]
  Y[1] = thoProj.orig[1]
  IF Y[1] < neg(M[1][0]) + neg(M[1][1]) + neg(M[1][2])
    RETURN FALSE
  END IF
  M[2][0] = -thoProj.comp[0][2]
  M[2][1] = -thoProj.comp[1][2]
  M[2][2] = -thoProj.comp[2][2]
  Y[2] = thoProj.orig[2]
  IF Y[2] < neg(M[2][0]) + neg(M[2][1]) + neg(M[2][2])
```

```
  RETURN FALSE
END IF
nbRows = 3
IF that.type == FrameCuboid
  M[nbRows][0] = thoProj.comp[0][0]
  M[nbRows][1] = thoProj.comp[1][0]
  M[nbRows][2] = thoProj.comp[2][0]
  Y[nbRows] = 1.0 - thoProj.orig[0]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
  M[nbRows][0] = thoProj.comp[0][1]
  M[nbRows][1] = thoProj.comp[1][1]
  M[nbRows][2] = thoProj.comp[2][1]
  Y[nbRows] = 1.0 - thoProj.orig[1]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
  M[nbRows][0] = thoProj.comp[0][2]
  M[nbRows][1] = thoProj.comp[1][2]
  M[nbRows][2] = thoProj.comp[2][2]
  Y[nbRows] = 1.0 - thoProj.orig[2]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
ELSE
  M[nbRows][0] =
    thoProj.comp[0][0] + thoProj.comp[0][1] + thoProj.comp[0][2]
  M[nbRows][1] =
    thoProj.comp[1][0] + thoProj.comp[1][1] + thoProj.comp[1][2]
  M[nbRows][2] =
    thoProj.comp[2][0] + thoProj.comp[2][1] + thoProj.comp[2][2]
  Y[nbRows] =
    1.0 - thoProj.orig[0] - thoProj.orig[1] - thoProj.orig[2]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
END
IF tho.type == FrameCuboid {
  M[nbRows][0] = 1.0
  M[nbRows][1] = 0.0
  M[nbRows][2] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = 1.0
  M[nbRows][2] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = 0.0
  M[nbRows][2] = 1.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
```

```
  ELSE
    M[nbRows][0] = 1.0
    M[nbRows][1] = 1.0
    M[nbRows][2] = 1.0
    Y[nbRows] = 1.0
    nbRows = nbRows + 1
  END
  M[nbRows][0] = -1.0
  M[nbRows][1] = 0.0
  M[nbRows][2] = 0.0
  Y[nbRows] = 0.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = -1.0
  M[nbRows][2] = 0.0
  Y[nbRows] = 0.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = 0.0
  M[nbRows][2] = -1.0
  Y[nbRows] = 0.0
  nbRows = nbRows + 1
  inconsistency =
    ElimVar3D(FST_VAR, M, Y, nbRows, 3, Mp, Yp, nbRowsP)
  IF inconsistency == TRUE
    RETURN FALSE
  END
  inconsistency =
    ElimVar3D(FST_VAR, Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
  IF inconsistency == TRUE
    RETURN FALSE
  END
  GetBound3D(THD_VAR, Mpp, Ypp, nbRowsPP, bdgBoxLocal)
  IF bdgBoxLocal.min[THD_VAR] >= bdgBoxLocal.max[THD_VAR]
    RETURN FALSE
  END
  ElimVar3D(SND_VAR, Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
  GetBound3D(SND_VAR, Mpp, Ypp, nbRowsPP, bdgBoxLocal)
  ElimVar3D(THD_VAR, M, Y, nbRows, 3, Mp, Yp, nbRowsP)
  ElimVar3D( SND_VAR, Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
  GetBound3D(FST_VAR, Mpp, Ypp, nbRowsPP, bdgBoxLocal)
  bdgBox = bdgBoxLocal
  RETURN TRUE
END

origP3D = [0.0, 0.0, 0.0]
compP3D = [
  [1.0, 0.0, 0.0],
  [0.0, 1.0, 0.0],
  [0.0, 0.0, 1.0]]
P3D = Frame3DCreateStatic(FrameTetrahedron, origP3D, compP3D)
origQ3D = [0.5, 0.5, 0.5]
compQ3D = [
  [2.0, 0.0, 0.0],
  [0.0, 2.0, 0.0],
  [0.0, 0.0, 2.0]]
Q3D = Frame3DCreateStatic(FrameTetrahedron, origQ3D, compQ3D)
isIntersecting3D = FMBTestIntersection3D(P3D, Q3D, bdgBox3DLocal)
IF isIntersecting3D == TRUE
  PRINT "Intersection detected."
  Frame3DExportBdgBox(Q3D, bdgBox3DLocal, bdgBox3D)
  AABB3DPrint(bdgBox3D)
```

```
ELSE
  PRINT "No intersection."
END IF
```

### 3.3 2D dynamic

```
ENUM FrameType
  FrameCuboid,
  FrameTetrahedron
END ENUM

STRUCT AABB2DTime
  // x,y,t
  real min[3]
  real max[3]
END STRUCT

STRUCT Frame2DTime
  FrameType type
  real orig[2]
  // comp[iComp][iAxis]
  real comp[2][2]
  AABB2DTime bdgBox
  real invComp[2][2]
  real speed[2]
END STRUCT

FUNCTION powi(base, exp)
    res = 1
    FOR i=0..(exp - 1)
      res = res * base
    END FOR
    RETURN res
END FUNCTION

FUNCTION Frame2DTimePrint(that)
  IF that.type == FrameTetrahedron
    PRINT "T"
  ELSE IF that.type == FrameCuboid
    PRINT "C"
  END IF
  PRINT "o("
  FOR i = 0..1
    PRINT that.orig[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  PRINT ") s("
  FOR i = 0..1
    PRINT that.speed[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  comp = ['x', 'y']
  FOR j = 0..1
    PRINT ") " comp[j] "("
    FOR i = 0..1
      PRINT that.comp[j][i]
      IF i < 1
```

```
      PRINT ","
    END IF
  END FOR
END FOR
PRINT ")"
END FUNCTION

FUNCTION AABB2DTimePrint(that)
  PRINT "minXYT("
  FOR i = 0..2
    PRINT that.min[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  PRINT ")-maxXYT("
  FOR i = 0..2
    PRINT that.max[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  PRINT ")"
END FUNCTION

FUNCTION Frame2DTimeExportBdgBox(that, bdgBox, bdgBoxProj)
  bdgBoxProj.min[2] = bdgBox.min[2]
  bdgBoxProj.max[2] = bdgBox.max[2]
  FOR i = 0..1
    bdgBoxProj.max[i] = that.orig[i] + that.speed[i] * bdgBox.min[2]
    FOR j = 0..1
      bdgBoxProj.max[i] =
        bdgBoxProj.max[i] + that.comp[j][i] * bdgBox.min[j]
    END FOR
    bdgBoxProj.min[i] = bdgBoxProj.max[i]
  END FOR
  nbVertices = powi(2, 2)
  FOR iVertex = 1..(nbVertices - 1)
    FOR i = 0..1
      IF (iVertex & (1 << i)) == TRUE
        v[i] = bdgBox.max[i]
      ELSE
        v[i] = bdgBox.min[i]
      END IF
    END FOR
    FOR i = 0..1
      w[i] = that.orig[i]
      FOR j = 0..1
        w[i] = w[i] + that.comp[j][i] * v[j]
      END FOR
    END FOR
    FOR i = 0..1
      IF bdgBoxProj.min[i] > w[i] + that.speed[i] * bdgBox.min[2]
        bdgBoxProj.min[i] = w[i] + that.speed[i] * bdgBox.min[2]
      END IF
      IF bdgBoxProj.min[i] > w[i] + that.speed[i] * bdgBox.max[2]
        bdgBoxProj.min[i] = w[i] + that.speed[i] * bdgBox.max[2]
      END IF
      IF bdgBoxProj.max[i] < w[i] + that.speed[i] * bdgBox.min[2]
        bdgBoxProj.max[i] = w[i] + that.speed[i] * bdgBox.min[2]
      END IF
      IF bdgBoxProj.max[i] < w[i] + that.speed[i] * bdgBox.max[2]
```

```
        bdgBoxProj.max[i] = w[i] + that.speed[i] * bdgBox.max[2]
      END IF
    END FOR
  END FOR
END FUNCTION

FUNCTION Frame2DTimeImPortFrame(P, Q, Qp)
  FOR i = 0..1
    v[i] = Q.orig[i] - P.orig[i]
    s[i] = Q.speed[i] - P.speed[i]
  END FOR
  FOR i = 0..1
    Qp.orig[i] = 0.0
    Qp.speed[i] = 0.0
    FOR j = 0..1
      Qp.orig[i] = Qp.orig[i] + P.invComp[j][i] * v[j]
      Qp.speed[i] = Qp.speed[i] + P.invComp[j][i] * s[j]
      Qp.comp[j][i] = 0.0
      FOR k = 0..1
        Qp.comp[j][i] = Qp.comp[j][i] + P.invComp[k][i] * Q.comp[j][k]
      END FOR
    END FOR
  END FOR
END FUNCTION

FUNCTION Frame2DTimeUpdateInv(that)
  det = that.comp[0][0] * that.comp[1][1] -
    that.comp[1][0] * that.comp[0][1]
  that.invComp[0][0] = that.comp[1][1] / det
  that.invComp[0][1] = -that.comp[0][1] / det
  that.invComp[1][0] = -that.comp[1][0] / det
  that.invComp[1][1] = that.comp[0][0] / det
END FUNCTION

FUNCTION Frame2DTimeCreateStatic(type, orig, comp)
  that.type = type
  FOR iAxis = 0..1
    that.orig[iAxis] = orig[iAxis]
    that.speed[iAxis] = speed[iAxis]
    FOR iComp = 0..1
      that.comp[iComp][iAxis] = comp[iComp][iAxis]
    END FOR
  END FOR
  FOR iAxis = 0..1
    min = orig[iAxis]
    max = orig[iAxis]
    FOR iComp = 0..1
      IF that.type == FrameCuboid
        IF that.comp[iComp][iAxis] < 0.0
          min += that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0
          max += that.comp[iComp][iAxis]
        END IF
      ELSE IF that.type == FrameTetrahedron
        IF that.comp[iComp][iAxis] < 0.0 AND
          min > orig[iAxis] + that.comp[iComp][iAxis]
          min = orig[iAxis] + that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0 AND
          max < orig[iAxis] + that.comp[iComp][iAxis]
          max = orig[iAxis] + that.comp[iComp][iAxis]
```

```
        END IF
      END IF
    END FOR
    IF that.speed[iAxis] < 0.0
      min = min + that.speed[iAxis]
    END IF
    IF that.speed[iAxis] > 0.0
      max = max + that.speed[iAxis]
    END IF
    that.bdgBox.min[iAxis] = min
    that.bdgBox.max[iAxis] = max
  END FOR
  that.bdgBox.min[2] = 0.0
  that.bdgBox.max[2] = 1.0
  Frame2DTimeUpdateInv(that)
  RETURN that
END FUNCTION

FUNCTION Sgn(v)
  IF 0.0 < v
    a = 1
  ELSE
    a = 0
  END IF
  IF v < 0.0
    b = 1
  ELSE
    b = 0
  END IF
  RETURN A - B
END FUNCTION

FUNCTION Neg(x)
  IF x < 0.0
    RETURN x
  ELSE
    RETURN 0.0
  END IF
END FUNCTION

FST_VAR = 0
SND_VAR = 1
THD_VAR = 2

FUNCTION ElimVar2DTime(iVar, M, Y, nbRows, nbCols, Mp, Yp, nbRemainRows)
  nbRemainRows = 0
  FOR iRow = 0..(nbRows - 2)
    IF M[iRow][iVar] <> 0.0
      FOR jRow = (iRow + 1)..(nbRows - 1)
        IF sgn(M[iRow][iVar]) <> sgn(M[jRow][iVar]) AND
            M[jRow][iVar] <> 0.0
          sumNegCoeff = 0.0
          jCol = 0
          FOR iCol = 0..(nbCols - 1)
            IF iCol <> iVar
              Mp[nbRemainRows][jCol] =
                M[iRow][iCol] / fabs(M[iRow][iVar]) +
                M[jRow][iCol] / fabs(M[jRow][iVar])
              sumNegCoeff = sumNegCoeff + neg(Mp[nbRemainRows][jCol])
              jCol = jCol + 1
            END IF
          END FOR
```

```
          Yp[nbRemainRows] =
            Y[iRow] / fabs(M[iRow][iVar]) +
            Y[jRow] / fabs(M[jRow][iVar])
          IF Yp[nbRemainRows] < sumNegCoeff
            RETURN TRUE
          END IF
          nbRemainRows = nbRemainRows + 1
        END IF
      END FOR
    END IF
  END FOR
  FOR iRow = 0..(nbRows - 1)
    IF M[iRow][iVar] == 0.0
      jCol = 0
      FOR iCol = 0..(nbCols - 1)
        IF iCol <> iVar
          Mp[nbRemainRows][jCol] = M[iRow][iCol]
          jCol = jCol + 1
        END IF
      END FOR
      Yp[nbRemainRows] = Y[iRow]
      nbRemainRows = nbRemainRows + 1
    END IF
  END FOR
  RETURN FALSE
END FUNCTION

FUNCTION GetBound2DTime(iVar, M, Y, nbRows, bdgBox)
  bdgBox.min[iVar] = 0.0
  bdgBox.max[iVar] = 1.0
  FOR jRow = 0..(nbRows - 1)
    IF M[jRow][0] > 0.0
      double y = Y[jRow] / M[jRow][0]
      IF bdgBox.max[iVar] > y
        bdgBox.max[iVar] = y
      END IF
    ELSE IF M[jRow][0] < 0.0
      double y = Y[jRow] / M[jRow][0]
      IF bdgBox.min[iVar] < y
        bdgBox.min[iVar] = y
      END IF
    END IF
  END FOR
END FUNCTION

FUNCTION FMBTestIntersection2DTime(that, tho, bdgBox)
  Frame2DTimeImportFrame(that, tho, &thoProj)
  M[0][0] = -thoProj.comp[0][0]
  M[0][1] = -thoProj.comp[1][0]
  M[0][2] = -thoProj.speed[0]
  Y[0] = thoProj.orig[0]
  IF (Y[0] < neg(M[0][0]) + neg(M[0][1]) + neg(M[0][2]))
    RETURN FALSE
  END IF
  M[1][0] = -thoProj.comp[0][1]
  M[1][1] = -thoProj.comp[1][1]
  M[1][2] = -thoProj.speed[1]
  Y[1] = thoProj.orig[1]
  IF (Y[1] < neg(M[1][0]) + neg(M[1][1]) + neg(M[1][2]))
    RETURN FALSE
  END IF
  nbRows = 2
```

```
IF that.type == FrameCuboid
  M[nbRows][0] = thoProj.comp[0][0]
  M[nbRows][1] = thoProj.comp[1][0]
  M[nbRows][2] = thoProj.speed[0]
  Y[nbRows] = 1.0 - thoProj.orig[0]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
  M[nbRows][0] = thoProj.comp[0][1]
  M[nbRows][1] = thoProj.comp[1][1]
  M[nbRows][2] = thoProj.speed[1]
  Y[nbRows] = 1.0 - thoProj.orig[1]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  nbRows = nbRows + 1
ELSE
  M[nbRows][0] = thoProj.comp[0][0] + thoProj.comp[0][1]
  M[nbRows][1] = thoProj.comp[1][0] + thoProj.comp[1][1]
  M[nbRows][2] = thoProj.speed[0] + thoProj.speed[1]
  Y[nbRows] = 1.0 - thoProj.orig[0] - thoProj.orig[1]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  nbRows = nbRows + 1
END
IF tho.type == FrameCuboid
  M[nbRows][0] = 1.0
  M[nbRows][1] = 0.0
  M[nbRows][2] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = 1.0
  M[nbRows][2] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
ELSE
  M[nbRows][0] = 1.0
  M[nbRows][1] = 1.0
  M[nbRows][2] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
END IF
M[nbRows][0] = -1.0
M[nbRows][1] = 0.0
M[nbRows][2] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = -1.0
M[nbRows][2] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = 1.0
Y[nbRows] = 1.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
```

```
  M[nbRows][1] = 0.0
  M[nbRows][2] = -1.0
  Y[nbRows] = 0.0
  nbRows = nbRows + 1
  inconsistency =
    ElimVar2DTime(FST_VAR, M, Y, nbRows, 3, Mp, Yp, nbRowsP)
  IF inconsistency == TRUE
    RETURN FALSE
  END IF
  inconsistency =
    ElimVar2DTime(FST_VAR, Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
  IF inconsistency == TRUE
    RETURN FALSE
  END IF
  GetBound2DTime(THD_VAR, Mpp, Ypp, nbRowsPP, bdgBoxLocal)
  IF bdgBoxLocal.min[THD_VAR] >= bdgBoxLocal.max[THD_VAR]
    RETURN FALSE
  END IF
  ElimVar2DTime(SND_VAR, Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
  GetBound2DTime(SND_VAR, Mpp, Ypp, nbRowsPP, bdgBoxLocal)
  ElimVar2DTime(THD_VAR, M, Y, nbRows, 3, Mp, Yp, nbRowsP)
  ElimVar2DTime(SND_VAR, Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
  GetBound2DTime(FST_VAR, Mpp, Ypp, nbRowsPP, bdgBoxLocal)
  bdgBox = bdgBoxLocal
  RETURN TRUE
END FUNCTION

origP2DTime = [0.0, 0.0]
speedP2DTime = [0.0, 0.0]
compP2DTime = [
  [1.0, 0.0],
  [0.0, 1.0]]
P2DTime =
  Frame2DTimeCreateStatic(
    FrameCuboid, origP2DTime, speedP2DTime, compP2DTime)
origQ2DTime = [0.0,0.0]
speedQ2DTime = [0.0,0.0]
compQ2DTime = [
  [1.0, 0.0],
  [0.0, 1.0]]
Q2DTime =
  Frame2DTimeCreateStatic(
    FrameCuboid, origQ2DTime, speedQ2DTime, compQ2DTime)
isIntersecting2DTime =
  FMBTestIntersection2DTime(P2DTime, Q2DTime, bdgBox2DTimeLocal)
IF isIntersecting2DTime == TRUE
  PRINT "Intersection detected."
  Frame2DTimeExportBdgBox(Q2DTime, bdgBox2DTimeLocal, bdgBox2DTime)
  AABB2DTimePrint(bdgBox2DTime)
ELSE
  PRINT "No intersection."
END IF
```

## 3.4 3D dynamic

```
ENUM FrameType
  FrameCuboid,
  FrameTetrahedron
END ENUM

STRUCT AABB3DTime
```

```
  // x,y,z,t
  real min[4]
  real max[4]
END STRUCT

STRUCT Frame3DTime
  FrameType type
  real orig[3]
  // comp[iComp][iAxis]
  real comp[3][3]
  AABB3DTime bdgBox
  real invComp[3][3]
  real speed[3]
END STRUCT

FUNCTION powi(base, exp)
    res = 1
    FOR i=0..(exp - 1)
      res = res * base
    END FOR
    RETURN res
END FUNCTION

FUNCTION Frame3DTimePrint(that)
  IF that.type == FrameTetrahedron
    PRINT "T"
  ELSE IF that.type == FrameCuboid
    PRINT "C"
  END IF
  PRINT "o("
  FOR (i = 0..2
    PRINT that.orig[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  PRINT " s("
  FOR i = 0..2
    PRINT that.speed[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  comp = ['x', 'y', 'z']
  FOR j = 0..2
    PRINT " " comp[j] "("
    FOR i = 0..2
      PRINT that.comp[j][i]
      IF i < 2
        PRINT ","
      END IF
    END FOR
  END FOR
  PRINT ""
END FUNCTION

FUNCTION AABB3DTimePrint(that)
  PRINT "minXYZT("
  FOR i = 0..3
    PRINT that.min[i]
    IF i < 3
      PRINT ","
```

```
    END IF
  END FOR
  PRINT ")-maxXYZT("
  FOR i = 0..3
    PRINT that.max[i]
    IF i < 3
      PRINT ","
    END IF
  END FOR
  PRINT ")"
END FUNCTION

FUNCTION Frame3DTimeExportBdgBox(that, bdgBox, bdgBoxProj)
  bdgBoxProj.min[3] = bdgBox.min[3]
  bdgBoxProj.max[3] = bdgBox.max[3]
  FOR i = 0..2
    bdgBoxProj.max[i] = that.orig[i] + that.speed[i] * bdgBox.min[3]
    FOR j = 0..2
      bdgBoxProj.max[i] =
        bdgBoxProj.max[i] + that.comp[j][i] * bdgBox.min[j]
    END FOR
    bdgBoxProj.min[i] = bdgBoxProj.max[i]
  END FOR
  nbVertices = powi(2, 3)
  FOR iVertex = 1..(nbVertices - 1)
    FOR i = 0..2
      IF (iVertex & (1 << i)) == TRUE
        v[i] = bdgBox.max[i]
      ELSE
        v[i] = bdgBox.min[i]
      END IF
    END FOR
    FOR i = 0..2
      w[i] = that.orig[i]
      FOR j = 0..2
        w[i] = w[i] + that.comp[j][i] * v[j]
      END FOR
    END FOR
    FOR i = 0..2
      IF bdgBoxProj.min[i] > w[i] + that.speed[i] * bdgBox.min[3]
        bdgBoxProj.min[i] = w[i] + that.speed[i] * bdgBox.min[3]
      END IF
      IF bdgBoxProj.min[i] > w[i] + that.speed[i] * bdgBox.max[3]
        bdgBoxProj.min[i] = w[i] + that.speed[i] * bdgBox.max[3]
      END IF
      IF bdgBoxProj.max[i] < w[i] + that.speed[i] * bdgBox.min[3]
        bdgBoxProj.max[i] = w[i] + that.speed[i] * bdgBox.min[3]
      END IF
      IF bdgBoxProj.max[i] < w[i] + that.speed[i] * bdgBox.max[3]
        bdgBoxProj.max[i] = w[i] + that.speed[i] * bdgBox.max[3]
      END IF
    END FOR
  END FOR
END FUNCTION

FUNCTION Frame3DTimeImPortFrame(P, Q, Qp)
  FOR i = 0..2
    v[i] = Q.orig[i] - P.orig[i]
    s[i] = Q.speed[i] - P.speed[i]
  END FOR
  FOR i = 0..2
    Qp.orig[i] = 0.0
```

```
    Qp.speed[i] = 0.0
    FOR j = 0..2
      Qp.orig[i] = Qp.orig[i] + P.invComp[j][i] * v[j]
      Qp.speed[i] = Qp.speed[i] + P.invComp[j][i] * s[j]
      Qp.comp[j][i] = 0.0
      FOR k = 0..2
        Qp.comp[j][i] = Qp.comp[j][i] + P.invComp[k][i] * Q.comp[j][k]
      END FOR
    END FOR
  END FOR
END FUNCTION

FUNCTION Frame3DTimeUpdateInv(that)
  det =
    that.comp[0][0] *
    (that.comp[1][1] * that.comp[2][2] - that.comp[1][2] * that.comp[2][1])
        -
    that.comp[1][0] *
    (that.comp[0][1] * that.comp[2][2] - that.comp[0][2] * that.comp[2][1])
        +
    that.comp[2][0] *
    (that.comp[0][1] * that.comp[1][2] - that.comp[0][2] * that.comp[1][1])
  that.invComp[0][0] = (that.comp[1][1] * that.comp[2][2] -
    that.comp[2][1] * that.comp[1][2]) / det
  that.invComp[0][1] = (that.comp[2][1] * that.comp[0][2] -
    that.comp[2][2] * that.comp[0][1]) / det
  that.invComp[0][2] = (that.comp[0][1] * that.comp[1][2] -
    that.comp[0][2] * that.comp[1][1]) / det
  that.invComp[1][0] = (that.comp[2][0] * that.comp[1][2] -
    that.comp[2][2] * that.comp[1][0]) / det
  that.invComp[1][1] = (that.comp[0][0] * that.comp[2][2] -
    that.comp[2][0] * that.comp[0][2]) / det
  that.invComp[1][2] = (that.comp[0][2] * that.comp[1][0] -
    that.comp[1][2] * that.comp[0][0]) / det
  that.invComp[2][0] = (that.comp[1][0] * that.comp[2][1] -
    that.comp[2][0] * that.comp[1][1]) / det
  that.invComp[2][1] = (that.comp[0][1] * that.comp[2][0] -
    that.comp[2][1] * that.comp[0][0]) / det
  that.invComp[2][2] = (that.comp[0][0] * that.comp[1][1] -
    that.comp[1][0] * that.comp[0][1]) / det
END FUNCTION

FUNCTION Frame3DTimeCreateStatic(type, orig, comp)
  that.type = type
  FOR iAxis = 0..2
    that.orig[iAxis] = orig[iAxis]
    that.speed[iAxis] = speed[iAxis]
    FOR iComp = 0..2
      that.comp[iComp][iAxis] = comp[iComp][iAxis]
    END FOR
  END FOR
  FOR iAxis = 0..2
    min = orig[iAxis]
    max = orig[iAxis]
    FOR iComp = 0..2
      IF that.type == FrameCuboid
        IF that.comp[iComp][iAxis] < 0.0
          min += that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0
          max += that.comp[iComp][iAxis]
        END IF
```

```
      ELSE IF that.type == FrameTetrahedron
        IF that.comp[iComp][iAxis] < 0.0 AND
          min > orig[iAxis] + that.comp[iComp][iAxis]
          min = orig[iAxis] + that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0 AND
          max < orig[iAxis] + that.comp[iComp][iAxis]
          max = orig[iAxis] + that.comp[iComp][iAxis]
        END IF
      END IF
    END FOR
    IF that.speed[iAxis] < 0.0
      min = min + that.speed[iAxis]
    END IF
    IF that.speed[iAxis] > 0.0
      max = max + that.speed[iAxis]
    END IF
    that.bdgBox.min[iAxis] = min
    that.bdgBox.max[iAxis] = max
  END FOR
  that.bdgBox.min[3] = 0.0
  that.bdgBox.max[3] = 1.0
  Frame3DTimeUpdateInv(that)
  RETURN that
END FUNCTION

FUNCTION Sgn(v)
  IF 0.0 < v
    a = 1
  ELSE
    a = 0
  END IF
  IF v < 0.0
    b = 1
  ELSE
    b = 0
  END IF
  RETURN A - B
END FUNCTION

FUNCTION Neg(x)
  IF x < 0.0
    RETURN x
  ELSE
    RETURN 0.0
  END IF
END FUNCTION

FST_VAR = 0
SND_VAR = 1
THD_VAR = 2
FOR_VAR = 3

FUNCTION ElimVar3DTime(iVar, M, Y, nbRows, nbCols, Mp, Yp, nbRemainRows)
  nbRemainRows = 0
  FOR iRow = 0..(nbRows - 2)
    IF M[iRow][iVar] <> 0.0
      FOR jRow = (iRow + 1)..(nbRows - 1)
        IF sgn(M[iRow][iVar]) <> sgn(M[jRow][iVar]) AND
            M[jRow][iVar] <> 0.0
          sumNegCoeff = 0.0
          jCol = 0
```

```
          FOR iCol = 0..(nbCols - 1)
            IF iCol <> iVar
              Mp[nbRemainRows][jCol] =
                M[iRow][iCol] / fabs(M[iRow][iVar]) +
                M[jRow][iCol] / fabs(M[jRow][iVar])
              sumNegCoeff = sumNegCoeff + neg(Mp[nbRemainRows][jCol])
              jCol = jCol + 1
            END IF
          END FOR
          Yp[nbRemainRows] =
            Y[iRow] / fabs(M[iRow][iVar]) +
            Y[jRow] / fabs(M[jRow][iVar])
          IF Yp[nbRemainRows] < sumNegCoeff
            RETURN TRUE
          END IF
          nbRemainRows = nbRemainRows + 1
        END IF
      END FOR
    END IF
  END FOR
  FOR iRow = 0..(nbRows - 1)
    IF M[iRow][iVar] == 0.0
      jCol = 0
      FOR iCol = 0..(nbCols - 1)
        IF iCol <> iVar
          Mp[nbRemainRows][jCol] = M[iRow][iCol]
          jCol = jCol + 1
        END IF
      END FOR
      Yp[nbRemainRows] = Y[iRow]
      nbRemainRows = nbRemainRows + 1
    END IF
  END FOR
  RETURN FALSE
END FUNCTION

FUNCTION GetBound3DTime(iVar, M, Y, nbRows, bdgBox)
  bdgBox.min[iVar] = 0.0
  bdgBox.max[iVar] = 1.0
  FOR jRow = 0..(nbRows - 1)
    IF M[jRow][0] > 0.0
      y = Y[jRow] / M[jRow][0]
      IF bdgBox.max[iVar] > y
        bdgBox.max[iVar] = y
      END IF
    ELSE IF M[jRow][0] < 0.0
      y = Y[jRow] / M[jRow][0]
      IF bdgBox.min[iVar] < y
        bdgBox.min[iVar] = y
      END IF
    END IF
  END FOR
END FUNCTION

FUNCTION FMBTestIntersection3DTime(that, tho, bdgBox)
  Frame3DTimeImportFrame(that, tho, thoProj)
  M[0][0] = -thoProj.comp[0][0]
  M[0][1] = -thoProj.comp[1][0]
  M[0][2] = -thoProj.comp[2][0]
  M[0][3] = -thoProj.speed[0]
  Y[0] = thoProj.orig[0]
  IF (Y[0] < neg(M[0][0]) + neg(M[0][1]) + neg(M[0][2]) + neg(M[0][3]))
```

```
  RETURN FALSE
END IF
M[1][0] = -thoProj.comp[0][1]
M[1][1] = -thoProj.comp[1][1]
M[1][2] = -thoProj.comp[2][1]
M[1][3] = -thoProj.speed[1]
Y[1] = thoProj.orig[1]
IF Y[1] < neg(M[1][0]) + neg(M[1][1]) + neg(M[1][2]) + neg(M[1][3])
  RETURN FALSE
END IF
M[2][0] = -thoProj.comp[0][2]
M[2][1] = -thoProj.comp[1][2]
M[2][2] = -thoProj.comp[2][2]
M[2][3] = -thoProj.speed[2]
Y[2] = thoProj.orig[2]
IF (Y[2] < neg(M[2][0]) + neg(M[2][1]) + neg(M[2][2]) + neg(M[2][3]))
  RETURN FALSE
nbRows = 3
IF that.type == FrameCuboid
  M[nbRows][0] = thoProj.comp[0][0]
  M[nbRows][1] = thoProj.comp[1][0]
  M[nbRows][2] = thoProj.comp[2][0]
  M[nbRows][3] = thoProj.speed[0]
  Y[nbRows] = 1.0 - thoProj.orig[0]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2]) + neg(M[nbRows][3])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
  M[nbRows][0] = thoProj.comp[0][1]
  M[nbRows][1] = thoProj.comp[1][1]
  M[nbRows][2] = thoProj.comp[2][1]
  M[nbRows][3] = thoProj.speed[1]
  Y[nbRows] = 1.0 - thoProj.orig[1]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2]) + neg(M[nbRows][3])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
  M[nbRows][0] = thoProj.comp[0][2]
  M[nbRows][1] = thoProj.comp[1][2]
  M[nbRows][2] = thoProj.comp[2][2]
  M[nbRows][3] = thoProj.speed[2]
  Y[nbRows] = 1.0 - thoProj.orig[2]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2]) + neg(M[nbRows][3])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
ELSE
  M[nbRows][0] =
    thoProj.comp[0][0] + thoProj.comp[0][1] + thoProj.comp[0][2]
  M[nbRows][1] =
    thoProj.comp[1][0] + thoProj.comp[1][1] + thoProj.comp[1][2]
  M[nbRows][2] =
    thoProj.comp[2][0] + thoProj.comp[2][1] + thoProj.comp[2][2]
  M[nbRows][3] = thoProj.speed[0] + thoProj.speed[1] + thoProj.speed[2]
  Y[nbRows] = 1.0 - thoProj.orig[0] - thoProj.orig[1] - thoProj.orig[2]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2]) + neg(M[nbRows][3])
    RETURN FALSE
  END IF
```

```
  nbRows = nbRows + 1
END IF
IF tho.type == FrameCuboid
  M[nbRows][0] = 1.0
  M[nbRows][1] = 0.0
  M[nbRows][2] = 0.0
  M[nbRows][3] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = 1.0
  M[nbRows][2] = 0.0
  M[nbRows][3] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = 0.0
  M[nbRows][2] = 1.0
  M[nbRows][3] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
ELSE
  M[nbRows][0] = 1.0
  M[nbRows][1] = 1.0
  M[nbRows][2] = 1.0
  M[nbRows][3] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
END IF
M[nbRows][0] = -1.0
M[nbRows][1] = 0.0
M[nbRows][2] = 0.0
M[nbRows][3] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = -1.0
M[nbRows][2] = 0.0
M[nbRows][3] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = -1.0
M[nbRows][3] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = 0.0
M[nbRows][3] = 1.0
Y[nbRows] = 1.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = 0.0
M[nbRows][3] = -1.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
inconsistency =
  ElimVar3DTime(FST_VAR, M, Y, nbRows, 4, Mp, Yp, nbRowsP)
IF inconsistency == TRUE
```

```
    RETURN FALSE
  END IF
  inconsistency =
    ElimVar3DTime(FST_VAR, Mp, Yp, nbRowsP, 3, Mpp, Ypp, nbRowsPP)
  IF inconsistency == TRUE
    RETURN FALSE
  END IF
  inconsistency =
    ElimVar3DTime(FST_VAR, Mpp, Ypp, nbRowsPP, 2, Mppp, Yppp, nbRowsPPP)
  IF inconsistency == TRUE
    RETURN FALSE
  END IF
  GetBound3DTime(FOR_VAR, Mppp, Yppp, nbRowsPPP, bdgBoxLocal)
  IF bdgBoxLocal.min[FOR_VAR] >= bdgBoxLocal.max[FOR_VAR]
    RETURN FALSE
  END IF
  ElimVar3DTime(SND_VAR, Mpp, Ypp, nbRowsPP, 2, Mppp, Yppp, nbRowsPPP)
  GetBound3DTime(THD_VAR, Mppp, Yppp, nbRowsPPP, bdgBoxLocal)
  ElimVar3DTime(FOR_VAR, M, Y, nbRows, 4, Mp, Yp, nbRowsP)
  ElimVar3DTime(THD_VAR, Mp, Yp, nbRowsP, 3, Mpp, Ypp, nbRowsPP)
  ElimVar3DTime(SND_VAR, Mpp, Ypp, nbRowsPP, 2, Mppp, Yppp, nbRowsPPP)
  GetBound3DTime(FST_VAR, Mppp, Yppp, nbRowsPPP, bdgBoxLocal)
  ElimVar3DTime(FST_VAR, Mpp, Ypp, nbRowsPP, 2, Mppp, Yppp, nbRowsPPP)
  GetBound3DTime(SND_VAR, Mppp, Yppp, nbRowsPPP, bdgBoxLocal)
  bdgBox = bdgBoxLocal
  RETURN TRUE
END FUNCTION

origP3DTime = [0.0, 0.0, 0.0]
speedP3DTime = [0.0, 0.0, 0.0]
compP3DTime = [
  [1.0, 0.0, 0.0],
  [0.0, 1.0, 0.0],
  [0.0, 0.0, 1.0]]
P3DTime =
  Frame3DTimeCreateStatic(
    FrameCuboid, origP3DTime, speedP3DTime, compP3DTime)
origQ3DTime = [0.0, 0.0, 0.0]
speedQ3DTime = [0.0, 0.0, 0.0]
compQ3DTime = [
  [1.0, 0.0, 0.0],
  [0.0, 1.0, 0.0],
  [0.0, 0.0, 1.0]]
Q3DTime =
  Frame3DTimeCreateStatic(
    FrameCuboid, origQ3DTime, speedQ3DTime, compQ3DTime)
isIntersecting3DTime =
  FMBTestIntersection3DTime(P3DTime, Q3DTime, bdgBox3DTimeLocal)
IF isIntersecting3DTime
  PRINT "Intersection detected."
  Frame3DTimeExportBdgBox(Q3DTime, bdgBox3DTimeLocal, bdgBox3DTime)
  AABB3DTimePrint(bdgBox3DTime)
ELSE
  PRINT "No intersection."
END IF
```

# 4 Implementation of the algorithms in C

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

## 4.1 Frames

### 4.1.1 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 {
  // x,y
  double min[2];
  double max[2];
} AABB2D;

typedef struct {
  // x,y,z
  double min[3];
  double max[3];
} AABB3D;

typedef struct {
  // x,y,t
  double min[3];
  double max[3];
} AABB2DTime;

typedef struct {
  // x,y,z,t
  double min[4];
  double max[4];
} AABB3DTime;

// Axis unaligned cuboid and tetrahedron structure
typedef struct {
  FrameType type;
  double orig[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 comp[3][3];
  // AABB of the frame
  AABB3D bdgBox;
  // Inverted components used during computation
  double invComp[3][3];
} Frame3D;

typedef struct {
  FrameType type;
  double orig[2];
  double comp[2][2];
  // AABB of the frame
  AABB2DTime bdgBox;
  // Inverted components used during computation
  double invComp[2][2];
  double speed[2];
} Frame2DTime;

typedef struct {
  FrameType type;
  double orig[3];
  double comp[3][3];
  // AABB of the frame
  AABB3DTime bdgBox;
  // Inverted components used during computation
  double invComp[3][3];
  double speed[3];
} Frame3DTime;

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

// Print the AABB 'that' on stdout
// Output format is
// (min[0], min[1], min[2], min[3])-(max[0], max[1], max[2], max[3])
void AABB2DPrint(const AABB2D* const that);
void AABB3DPrint(const AABB3D* const that);
void AABB2DTimePrint(const AABB2DTime* const that);
void AABB3DTimePrint(const AABB3DTime* 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])
// (speed[0], speed[1], speed[2])
void Frame2DPrint(const Frame2D* const that);
void Frame3DPrint(const Frame3D* const that);
void Frame2DTimePrint(const Frame2DTime* const that);
void Frame3DTimePrint(const Frame3DTime* 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]);
Frame2DTime Frame2DTimeCreateStatic(
  const FrameType type,
     const double orig[2],
     const double speed[2],
     const double comp[2][2]);
Frame3DTime Frame3DTimeCreateStatic(
  const FrameType type,
     const double orig[3],
     const double speed[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);
void Frame2DTimeImportFrame(
  const Frame2DTime* const P,
  const Frame2DTime* const Q,
        Frame2DTime* const Qp);
void Frame3DTimeImportFrame(
  const Frame3DTime* const P,
  const Frame3DTime* const Q,
        Frame3DTime* const Qp);

// 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);
void Frame3DExportBdgBox(
  const Frame3D* const that,
   const AABB3D* const bdgBox,
         AABB3D* const bdgBoxProj);
void Frame2DTimeExportBdgBox(
  const Frame2DTime* const that,
   const AABB2DTime* const bdgBox,
         AABB2DTime* const bdgBoxProj);
void Frame3DTimeExportBdgBox(
  const Frame3DTime* const that,
   const AABB3DTime* const bdgBox,
         AABB3DTime* const bdgBoxProj);

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

#endif
```

### 4.1.2 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);
void Frame2DTimeUpdateInv(Frame2DTime* const that);
void Frame3DTimeUpdateInv(Frame3DTime* 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;

}

Frame2DTime Frame2DTimeCreateStatic(
  const FrameType type,
     const double orig[2],
     const double speed[2],
```

```
   const double comp[2][2]) {

// Create the new Frame
Frame2DTime that;
that.type = type;
for (int iAxis = 2;
     iAxis--;) {

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

      }

    }
```

```
    }

    if (that.speed[iAxis] < 0.0) {

      min += that.speed[iAxis];

    }

    if (that.speed[iAxis] > 0.0) {

      max += that.speed[iAxis];

    }

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

  }

  that.bdgBox.min[2] = 0.0;
  that.bdgBox.max[2] = 1.0;


  // Calculate the inverse matrix
  Frame2DTimeUpdateInv(&that);

  // Return the new Frame
  return that;

}

Frame3DTime Frame3DTimeCreateStatic(
  const FrameType type,
     const double orig[3],
     const double speed[3],
     const double comp[3][3]) {

  // Create the new Frame
  Frame3DTime that;
  that.type = type;
  for (int iAxis = 3;
       iAxis--;) {

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

      }

    }

  }

  if (that.speed[iAxis] < 0.0) {

    min += that.speed[iAxis];

  }

  if (that.speed[iAxis] > 0.0) {

    max += that.speed[iAxis];

  }

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

}

that.bdgBox.min[3] = 0.0;
that.bdgBox.max[3] = 1.0;

// Calculate the inverse matrix
Frame3DTimeUpdateInv(&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] = -tc[0][1] / det;
  tic[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;

}

// Update the inverse components of the Frame 'that'
void Frame2DTimeUpdateInv(Frame2DTime* 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] = -tc[0][1] / det;
  tic[1][0] = -tc[1][0] / det;
  tic[1][1] = tc[0][0] / det;

}

void Frame3DTimeUpdateInv(Frame3DTime* 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];

      }
    }
  }
}

void Frame2DTimeImportFrame(
  const Frame2DTime* const P,
  const Frame2DTime* const Q,
        Frame2DTime* const Qp) {

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

  const double*  qs  = Q->speed;
        double*  qps = Qp->speed;
  const double*  ps  = P->speed;

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

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

    v[i] = qo[i] - po[i];
    s[i] = qs[i] - ps[i];

  }

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

    qpo[i] = 0.0;
    qps[i] = 0.0;

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

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

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

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

      }
    }
  }
}

void Frame3DTimeImportFrame(
  const Frame3DTime* const P,
  const Frame3DTime* const Q,
```

```
        Frame3DTime* const Qp) {

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

  const double*  qs  = Q->speed;
        double*  qps = Qp->speed;
  const double*  ps  = P->speed;

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

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

    v[i] = qo[i] - po[i];
    s[i] = qs[i] - ps[i];

  }

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

    qpo[i] = 0.0;
    qps[i] = 0.0;

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

      qpo[i] += pi[j][i] * v[j];
      qps[i] += pi[j][i] * s[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 (bbpmi[i] > w[i]) {

        bbpmi[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* bbpmi = 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];

    }

    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, 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];

      }
    }
  }

}

void Frame2DTimeExportBdgBox(
  const Frame2DTime* const that,
   const AABB2DTime* const bdgBox,
         AABB2DTime* const bdgBoxProj) {

  // Shortcuts
  const double* to    = that->orig;
  const double* ts    = that->speed;
  const double* bbmi  = bdgBox->min;
  const double* bbma  = bdgBox->max;
        double* bbpmi = bdgBoxProj->min;
        double* bbpma = bdgBoxProj->max;
  const double (*tc)[2] = that->comp;

  // The time component is not affected
  bbpmi[2] = bbmi[2];
  bbpma[2] = bbma[2];

  // 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] + ts[i] * bbmi[2];

  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 (bbpmi[i] > w[i] + ts[i] * bbmi[2]) {

      bbpmi[i] = w[i] + ts[i] * bbmi[2];

    }
```

```
      if (bbpmi[i] > w[i] + ts[i] * bbma[2]) {

        bbpmi[i] = w[i] + ts[i] * bbma[2];

      }
      if (bbpma[i] < w[i] + ts[i] * bbmi[2]) {

        bbpma[i] = w[i] + ts[i] * bbmi[2];

      }
      if (bbpma[i] < w[i] + ts[i] * bbma[2]) {

        bbpma[i] = w[i] + ts[i] * bbma[2];

      }
    }
  }

}

void Frame3DTimeExportBdgBox(
  const Frame3DTime* const that,
   const AABB3DTime* const bdgBox,
         AABB3DTime* const bdgBoxProj) {

  // Shortcuts
  const double* to    = that->orig;
  const double* ts    = that->speed;
  const double* bbmi  = bdgBox->min;
  const double* bbma  = bdgBox->max;
        double* bbpmi = bdgBoxProj->min;
        double* bbpma = bdgBoxProj->max;
  const double (*tc)[3] = that->comp;

  // The time component is not affected
  bbpmi[3] = bbmi[3];
  bbpma[3] = bbma[3];

  // 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] + ts[i] * bbmi[3];

    for (int j = 3;
         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, 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] + ts[i] * bbmi[3]) {

        bbpmi[i] = w[i] + ts[i] * bbmi[3];

      }
      if (bbpmi[i] > w[i] + ts[i] * bbma[3]) {

        bbpmi[i] = w[i] + ts[i] * bbma[3];

      }
      if (bbpma[i] < w[i] + ts[i] * bbmi[3]) {

        bbpma[i] = w[i] + ts[i] * bbmi[3];

      }
      if (bbpma[i] < w[i] + ts[i] * bbma[3]) {

        bbpma[i] = w[i] + ts[i] * bbma[3];

      }
    }
  }

}

// Print the AABB 'that' on stdout
// Output format is (min[0], min[1], ...)-(max[0], max[1], ...)
```

```
void AABB2DPrint(const AABB2D* const that) {

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

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

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

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

  }
  printf(")");

}

void AABB3DPrint(const AABB3D* const that) {

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

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

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

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

  }
  printf(")");

}

void AABB2DTimePrint(const AABB2DTime* const that) {

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

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

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

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

  }
  printf(")");

}

void AABB3DTimePrint(const AABB3DTime* const that) {

  printf("minXYZT(");
  for (int i = 0;
       i < 4;
       ++i) {

    printf("%f", that->min[i]);
    if (i < 3)
      printf(",");

  }
  printf(")-maxXYZT(");
  for (int i = 0;
       i < 4;
       ++i) {

    printf("%f", that->max[i]);
    if (i < 3)
      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("o(");
  for (int i = 0;
       i < 2;
       ++i) {

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

```
  }
  char comp[2] = {'x', 'y'};
  for (int j = 0;
       j < 2;
       ++j) {
    printf(") %c(", comp[j]);
    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("o(");
  for (int i = 0;
       i < 3;
       ++i) {

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

  }
  char comp[3] = {'x', 'y', 'z'};
  for (int j = 0;
       j < 3;
       ++j) {
    printf(") %c(", comp[j]);
    for (int i = 0;
         i < 3;
         ++i) {

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

    }
  }
  printf(")");

}

void Frame2DTimePrint(const Frame2DTime* const that) {
  if (that->type == FrameTetrahedron) {
    printf("T");
  } else if (that->type == FrameCuboid) {
    printf("C");
  }
  printf("o(");
```

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

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

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

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

  }
  char comp[2] = {'x', 'y'};
  for (int j = 0;
       j < 2;
       ++j) {
    printf(") %c(", comp[j]);
    for (int i = 0;
         i < 2;
         ++i) {

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

    }
  }
  printf(")");

}

void Frame3DTimePrint(const Frame3DTime* const that) {
  if (that->type == FrameTetrahedron) {
    printf("T");
  } else if (that->type == FrameCuboid) {
    printf("C");
  }
  printf("o(");
  for (int i = 0;
       i < 3;
       ++i) {

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

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

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

```
  }
  char comp[3] = {'x', 'y', 'z'};
  for (int j = 0;
       j < 3;
       ++j) {
    printf(") %c(", comp[j]);
    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;
}
```

## 4.2 FMB

### 4.2.1 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)
```

```
// The resulting AABB is given in 'tho' 's local coordinates system
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))

// Return x if x is negative, 0.0 else
#define neg(x) (x < 0.0 ? x : 0.0)

#define FST_VAR 0
#define SND_VAR 1

#define EPSILON 0.0000001

// ------------ 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 -------------

// 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
  int nbResRows = 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
    double fabsMIRowIVar = fabs(M[iRow][iVar]);

    // If the coefficient for the eliminated vairable is not null
    // in this row
    if (fabsMIRowIVar > EPSILON) {

      // Shortcuts
      int sgnMIRowIVar = sgn(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]) &&
            fabs(M[jRow][iVar]) > EPSILON) {

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

          // 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[nbResRows][jCol] =
```

```
              M[iRow][iCol] / fabsMIRowIVar +
              M[jRow][iCol] / fabs(M[jRow][iVar]);

            // Update the sum of the negative coefficient
            sumNegCoeff += neg(Mp[nbResRows][jCol]);

            // Increment the number of columns in the new inequality
            ++jCol;

          }

        }

        // Update the right side of the inequality
        Yp[nbResRows] =
          YIRowDivideByFabsMIRowIVar +
          Y[jRow] / fabs(M[jRow][iVar]);

        // If the right side of the inequality if lower than the sum of
        // negative coefficients in the row
        // (Add epsilon for numerical imprecision)
        if (Yp[nbResRows] < sumNegCoeff - EPSILON) {

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

        }

        // Increment the nb of rows into the result system
        ++nbResRows;

      }

    }

  }

}

// 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(MiRow[iVar]) < EPSILON) {

    // Shortcut
    double* MpnbResRows = Mp[nbResRows];

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

```
        if (iCol != iVar) {

          MpnbResRows[jCol] = MiRow[iCol];

          ++jCol;

        }

      }

      Yp[nbResRows] = Y[iRow];

      // Increment the nb of rows into the result system
      ++nbResRows;

    }

  }

  // Memorize the number of rows in the result system
  *nbRemainRows = nbResRows;

  // If we reach here the system is not inconsistent
  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 GetBound2D(
     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 < -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)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection2D(
  const Frame2D* const that,
  const Frame2D* const tho,
         AABB2D* const bdgBox) {
//Frame2DPrint(that);printf("\n");
//Frame2DPrint(tho);printf("\n");
  // 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[8][2];
  double Y[8];

  // Create the inequality system

  // -sum_iC_j,iX_i<=O_j
```

```
M[0][0] = -thoProj.comp[0][0];
M[0][1] = -thoProj.comp[1][0];
Y[0] = thoProj.orig[0];
if (Y[0] < neg(M[0][0]) + neg(M[0][1]))
  return false;

M[1][0] = -thoProj.comp[0][1];
M[1][1] = -thoProj.comp[1][1];
Y[1] = thoProj.orig[1];
if (Y[1] < neg(M[1][0]) + neg(M[1][1]))
  return false;

// Variable to memorise the nb of rows in the system
int nbRows = 2;

if (that->type == FrameCuboid) {

  // sum_iC_j,iX_i<=1.0-O_j
  M[nbRows][0] = thoProj.comp[0][0];
  M[nbRows][1] = thoProj.comp[1][0];
  Y[nbRows] = 1.0 - thoProj.orig[0];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]))
    return false;
  ++nbRows;

  M[nbRows][0] = thoProj.comp[0][1];
  M[nbRows][1] = thoProj.comp[1][1];
  Y[nbRows] = 1.0 - thoProj.orig[1];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]))
    return false;
  ++nbRows;

} else {

  // sum_j(sum_iC_j,iX_i)<=1.0-sum_iO_i
  M[nbRows][0] = thoProj.comp[0][0] + thoProj.comp[0][1];
  M[nbRows][1] = thoProj.comp[1][0] + thoProj.comp[1][1];
  Y[nbRows] = 1.0 - thoProj.orig[0] - thoProj.orig[1];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]))
    return false;
  ++nbRows;

}

if (tho->type == FrameCuboid) {

  // X_i <= 1.0
  M[nbRows][0] = 1.0;
  M[nbRows][1] = 0.0;
  Y[nbRows] = 1.0;
  ++nbRows;

  M[nbRows][0] = 0.0;
  M[nbRows][1] = 1.0;
  Y[nbRows] = 1.0;
  ++nbRows;

} else {

  // sum_iX_i<=1.0
  M[nbRows][0] = 1.0;
  M[nbRows][1] = 1.0;
```

```
  Y[nbRows] = 1.0;
  ++nbRows;

}

// -X_i <= 0.0
M[nbRows][0] = -1.0;
M[nbRows][1] = 0.0;
Y[nbRows] = 0.0;
++nbRows;

M[nbRows][0] = 0.0;
M[nbRows][1] = -1.0;
Y[nbRows] = 0.0;
++nbRows;

// 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 size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mp[24][2];
//double Yp[24];
double Mp[11][2];
double Yp[11];
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 inconsistent
```

```
  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
  // No need to check for consistency because we already know here
  // that the Frames are intersecting and the system is consistent
  inconsistency =
    ElimVar2D(
      SND_VAR,
      M,
      Y,
      nbRows,
      2,
      Mp,
      Yp,
      &nbRowsP);

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

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

    // Memorize the result
    *bdgBox = bdgBoxLocal;

  }

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

}
```

### 4.2.2 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)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection3D(
  const Frame3D* const that,
  const Frame3D* const tho,
         AABB3D* 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))

// Return x if x is negative, 0.0 else
#define neg(x) (x < 0.0 ? x : 0.0)

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

#define EPSILON 0.0000001

// ------------ 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
// 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 -------------

// 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
  int nbResRows = 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
    double fabsMIRowIVar = fabs(M[iRow][iVar]);

    // If the coefficient for the eliminated vairable is not null
    // in this row
    if (fabsMIRowIVar > EPSILON) {

      // Shortcuts
      int sgnMIRowIVar = sgn(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]) &&
            fabs(M[jRow][iVar]) > EPSILON) {

          // Declare a variable to memorize the sum of the negative
          // coefficients in the row
```

```
        double sumNegCoeff = 0.0;

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

            // Update the sum of the negative coefficient
            sumNegCoeff += neg(Mp[nbResRows][jCol]);

            // Increment the number of columns in the new inequality
            ++jCol;

          }

        }

        // Update the right side of the inequality
        Yp[nbResRows] =
          YIRowDivideByFabsMIRowIVar +
          Y[jRow] / fabs(M[jRow][iVar]);

        // If the right side of the inequality if lower than the sum of
        // negative coefficients in the row
        // (Add epsilon for numerical imprecision)
        if (Yp[nbResRows] < sumNegCoeff - EPSILON) {

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

        }

        // Increment the nb of rows into the result system
        ++nbResRows;

      }

    }

  }

}

// 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(MiRow[iVar]) < EPSILON) {

      // Shortcut
      double* MpnbResRows = Mp[nbResRows];

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

        if (iCol != iVar) {

          MpnbResRows[jCol] = MiRow[iCol];

          ++jCol;

        }

      }

      Yp[nbResRows] = Y[iRow];

      // Increment the nb of rows into the result system
      ++nbResRows;

    }

  }

  // Memorize the number of rows in the result system
  *nbRemainRows = nbResRows;

  // If we reach here the system is not inconsistent
  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 < -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)
// The resulting AABB is given in 'tho' 's local coordinates system
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 <= Y
// (M arrangement is [iRow][iCol])
double M[12][3];
double Y[12];

// Create the inequality system

// -sum_iC_j,iX_i<=O_j
M[0][0] = -thoProj.comp[0][0];
M[0][1] = -thoProj.comp[1][0];
M[0][2] = -thoProj.comp[2][0];
Y[0] = thoProj.orig[0];
if (Y[0] < neg(M[0][0]) + neg(M[0][1]) + neg(M[0][2]))
  return false;

M[1][0] = -thoProj.comp[0][1];
M[1][1] = -thoProj.comp[1][1];
M[1][2] = -thoProj.comp[2][1];
Y[1] = thoProj.orig[1];
if (Y[1] < neg(M[1][0]) + neg(M[1][1]) + neg(M[1][2]))
  return false;

M[2][0] = -thoProj.comp[0][2];
M[2][1] = -thoProj.comp[1][2];
M[2][2] = -thoProj.comp[2][2];
Y[2] = thoProj.orig[2];
if (Y[2] < neg(M[2][0]) + neg(M[2][1]) + neg(M[2][2]))
  return false;

// Variable to memorise the nb of rows in the system
int nbRows = 3;

if (that->type == FrameCuboid) {

  // sum_iC_j,iX_i<=1.0-O_j
  M[nbRows][0] = thoProj.comp[0][0];
  M[nbRows][1] = thoProj.comp[1][0];
  M[nbRows][2] = thoProj.comp[2][0];
  Y[nbRows] = 1.0 - thoProj.orig[0];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]))
    return false;
  ++nbRows;

  M[nbRows][0] = thoProj.comp[0][1];
  M[nbRows][1] = thoProj.comp[1][1];
  M[nbRows][2] = thoProj.comp[2][1];
  Y[nbRows] = 1.0 - thoProj.orig[1];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]))
    return false;
  ++nbRows;

  M[nbRows][0] = thoProj.comp[0][2];
  M[nbRows][1] = thoProj.comp[1][2];
  M[nbRows][2] = thoProj.comp[2][2];
  Y[nbRows] = 1.0 - thoProj.orig[2];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
```

```
                  neg(M[nbRows][2]))
    return false;
  ++nbRows;

} else {

  // sum_j(sum_iC_j,iX_i)<=1.0-sum_iO_i
  M[nbRows][0] =
    thoProj.comp[0][0] + thoProj.comp[0][1] + thoProj.comp[0][2];
  M[nbRows][1] =
    thoProj.comp[1][0] + thoProj.comp[1][1] + thoProj.comp[1][2];
  M[nbRows][2] =
    thoProj.comp[2][0] + thoProj.comp[2][1] + thoProj.comp[2][2];
  Y[nbRows] =
    1.0 - thoProj.orig[0] - thoProj.orig[1] - thoProj.orig[2];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]))
    return false;
  ++nbRows;

}

if (tho->type == FrameCuboid) {

  // X_i <= 1.0
  M[nbRows][0] = 1.0;
  M[nbRows][1] = 0.0;
  M[nbRows][2] = 0.0;
  Y[nbRows] = 1.0;
  ++nbRows;

  M[nbRows][0] = 0.0;
  M[nbRows][1] = 1.0;
  M[nbRows][2] = 0.0;
  Y[nbRows] = 1.0;
  ++nbRows;

  M[nbRows][0] = 0.0;
  M[nbRows][1] = 0.0;
  M[nbRows][2] = 1.0;
  Y[nbRows] = 1.0;
  ++nbRows;

} else {

  // sum_iX_i<=1.0
  M[nbRows][0] = 1.0;
  M[nbRows][1] = 1.0;
  M[nbRows][2] = 1.0;
  Y[nbRows] = 1.0;
  ++nbRows;

}

// -X_i <= 0.0
M[nbRows][0] = -1.0;
M[nbRows][1] = 0.0;
M[nbRows][2] = 0.0;
Y[nbRows] = 0.0;
++nbRows;

M[nbRows][0] = 0.0;
```

```
M[nbRows][1] = -1.0;
M[nbRows][2] = 0.0;
Y[nbRows] = 0.0;
++nbRows;

M[nbRows][0] = 0.0;
M[nbRows][1] = 0.0;
M[nbRows][2] = -1.0;
Y[nbRows] = 0.0;
++nbRows;

// 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 size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mp[48][3];
//double Yp[48];
double Mp[20][3];
double Yp[20];
int nbRowsP;

// Eliminate the first variable in the original system
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 size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mpp[624][3];
//double Ypp[624];
double Mpp[55][3];
double Ypp[55];
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 inconstent
if (bdgBoxLocal.min[THD_VAR] >= bdgBoxLocal.max[THD_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;

}

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

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

```
  // Now starts again from the initial systems and eliminate the
  // second and third variables to get the bounds of the first variable
  // No need to check for consistency because we already know here
  // that the Frames are intersecting and the system is consistent
  inconsistency =
    ElimVar3D(
      THD_VAR,
      M,
      Y,
      nbRows,
      3,
      Mp,
      Yp,
      &nbRowsP);

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

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

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

    // Memorize the result
    *bdgBox = bdgBoxLocal;

  }

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

}
```

### 4.2.3 2D dynamic

Header

```
#ifndef __FMB2DT_H_
#define __FMB2DT_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)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection2DTime(
  const Frame2DTime* const that,
  const Frame2DTime* const tho,
         AABB2DTime* 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))

// Return x if x is negative, 0.0 else
#define neg(x) (x < 0.0 ? x : 0.0)

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

#define EPSILON 0.0000001

// ------------ 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)[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 GetBound2DTime(
     const int iVar,
```

```
  const double (*M)[3],
  const double* Y,
     const int nbRows,
   AABB2DTime* const bdgBox);

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

// 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)[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
  int nbResRows = 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
    double fabsMIRowIVar = fabs(M[iRow][iVar]);

    // If the coefficient for the eliminated vairable is not null
    // in this row
    if (fabsMIRowIVar > EPSILON) {

      // Shortcuts
      int sgnMIRowIVar = sgn(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]) &&
            fabs(M[jRow][iVar]) > EPSILON) {

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

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

              // Update the sum of the negative coefficient
              sumNegCoeff += neg(Mp[nbResRows][jCol]);

              // Increment the number of columns in the new inequality
              ++jCol;

            }

          }

          // Update the right side of the inequality
          Yp[nbResRows] =
            YIRowDivideByFabsMIRowIVar +
            Y[jRow] / fabs(M[jRow][iVar]);

          // If the right side of the inequality if lower than the sum of
          // negative coefficients in the row
          // (Add epsilon for numerical imprecision)
          if (Yp[nbResRows] < sumNegCoeff - EPSILON) {

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

          }

          // Increment the nb of rows into the result system
          ++nbResRows;

        }

      }

    }

  }

  // 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(MiRow[iVar]) < EPSILON) {
```

```
      // Shortcut
      double* MpnbResRows = Mp[nbResRows];

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

        if (iCol != iVar) {

          MpnbResRows[jCol] = MiRow[iCol];

          ++jCol;

        }

      }

      Yp[nbResRows] = Y[iRow];

      // Increment the nb of rows into the result system
      ++nbResRows;

    }

  }

  // Memorize the number of rows in the result system
  *nbRemainRows = nbResRows;

  // If we reach here the system is not inconsistent
  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)[3],
  const double* Y,
     const int nbRows,
   AABB2DTime* 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 < -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)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection2DTime(
  const Frame2DTime* const that,
  const Frame2DTime* const tho,
         AABB2DTime* const bdgBox) {

  // Get the projection of the Frame 'tho' in Frame 'that' coordinates
  // system
  Frame2DTime thoProj;
  Frame2DTimeImportFrame(that, tho, &thoProj);

  // Declare two variables to memorize the system to be solved M.X <= Y
```

```
// (M arrangement is [iRow][iCol])
double M[10][3];
double Y[10];

// Create the inequality system

// -V_jT-sum_iC_j,iX_i<=O_j
M[0][0] = -thoProj.comp[0][0];
M[0][1] = -thoProj.comp[1][0];
M[0][2] = -thoProj.speed[0];
Y[0] = thoProj.orig[0];
if (Y[0] < neg(M[0][0]) + neg(M[0][1]) + neg(M[0][2]))
  return false;

M[1][0] = -thoProj.comp[0][1];
M[1][1] = -thoProj.comp[1][1];
M[1][2] = -thoProj.speed[1];
Y[1] = thoProj.orig[1];
if (Y[1] < neg(M[1][0]) + neg(M[1][1]) + neg(M[1][2]))
  return false;

// Variable to memorise the nb of rows in the system
int nbRows = 2;

if (that->type == FrameCuboid) {

  // V_jT+sum_iC_j,iX_i<=1.0-O_j
  M[nbRows][0] = thoProj.comp[0][0];
  M[nbRows][1] = thoProj.comp[1][0];
  M[nbRows][2] = thoProj.speed[0];
  Y[nbRows] = 1.0 - thoProj.orig[0];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]))
    return false;
  ++nbRows;

  M[nbRows][0] = thoProj.comp[0][1];
  M[nbRows][1] = thoProj.comp[1][1];
  M[nbRows][2] = thoProj.speed[1];
  Y[nbRows] = 1.0 - thoProj.orig[1];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]))
    return false;
  ++nbRows;

} else {

  // sum_j(V_jT+sum_iC_j,iX_i)<=1.0-sum_iO_i
  M[nbRows][0] = thoProj.comp[0][0] + thoProj.comp[0][1];
  M[nbRows][1] = thoProj.comp[1][0] + thoProj.comp[1][1];
  M[nbRows][2] = thoProj.speed[0] + thoProj.speed[1];
  Y[nbRows] = 1.0 - thoProj.orig[0] - thoProj.orig[1];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]))
    return false;
  ++nbRows;

}

if (tho->type == FrameCuboid) {

  // X_i <= 1.0
```

```
    M[nbRows][0] = 1.0;
    M[nbRows][1] = 0.0;
    M[nbRows][2] = 0.0;
    Y[nbRows] = 1.0;
    ++nbRows;

    M[nbRows][0] = 0.0;
    M[nbRows][1] = 1.0;
    M[nbRows][2] = 0.0;
    Y[nbRows] = 1.0;
    ++nbRows;

  } else {

    // sum_iX_i <=1.0
    M[nbRows][0] = 1.0;
    M[nbRows][1] = 1.0;
    M[nbRows][2] = 0.0;
    Y[nbRows] = 1.0;
    ++nbRows;

  }

  // -X_i <= 0.0
  M[nbRows][0] = -1.0;
  M[nbRows][1] = 0.0;
  M[nbRows][2] = 0.0;
  Y[nbRows] = 0.0;
  ++nbRows;

  M[nbRows][0] = 0.0;
  M[nbRows][1] = -1.0;
  M[nbRows][2] = 0.0;
  Y[nbRows] = 0.0;
  ++nbRows;

  // 0.0 <= t <= 1.0
  M[nbRows][0] = 0.0;
  M[nbRows][1] = 0.0;
  M[nbRows][2] = 1.0;
  Y[nbRows] = 1.0;
  ++nbRows;

  M[nbRows][0] = 0.0;
  M[nbRows][1] = 0.0;
  M[nbRows][2] = -1.0;
  Y[nbRows] = 0.0;
  ++nbRows;

  // Solve the system

  // Declare a AABB to memorize the bounding box of the intersection
  // in the coordinates system of that
  AABB2DTime bdgBoxLocal;

  // Declare variables to eliminate the first variable
  // The size of the array given in the doc is a majoring value.
  // Instead I use a smaller value which has proven to be sufficient
  // during tests, validation and qualification, to avoid running
  // into the heap limit and to optimize slightly the performance
  //double Mp[35][3];
  //double Yp[35];
```

```
double Mp[13][3];
double Yp[13];
int nbRowsP;

// Eliminate the first variable in the original system
bool inconsistency =
  ElimVar2DTime(
    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 size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mpp[342][3];
//double Ypp[342];
double Mpp[21][3];
double Ypp[21];
int nbRowsPP;

// Eliminate the second variable (which is the first in the new system)
inconsistency =
  ElimVar2DTime(
    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
GetBound2DTime(
  THD_VAR,
  Mpp,
  Ypp,
  nbRowsPP,
  &bdgBoxLocal);
```

```
// If the bounds are inconstent
if (bdgBoxLocal.min[THD_VAR] >= bdgBoxLocal.max[THD_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;

}

// Eliminate the third variable (which is the second in the new
// system)
inconsistency =
  ElimVar2DTime(
    SND_VAR,
    Mp,
    Yp,
    nbRowsP,
    2,
    Mpp,
    Ypp,
    &nbRowsPP);

// Get the bounds for the remaining second variable
GetBound2DTime(
  SND_VAR,
  Mpp,
  Ypp,
  nbRowsPP,
  &bdgBoxLocal);

// Now starts again from the initial systems and eliminate the
// second and third variables to get the bounds of the first variable
// No need to check for consistency because we already know here
// that the Frames are intersecting and the system is consistent
inconsistency =
  ElimVar2DTime(
    THD_VAR,
    M,
    Y,
    nbRows,
    3,
    Mp,
    Yp,
    &nbRowsP);

inconsistency =
  ElimVar2DTime(
    SND_VAR,
    Mp,
    Yp,
    nbRowsP,
    2,
    Mpp,
    Ypp,
```

```
      &nbRowsPP);

  GetBound2DTime(
    FST_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

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

    // Memorize the result
    *bdgBox = bdgBoxLocal;

  }

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

}
```

### 4.2.4 3D dynamic

Header

```
#ifndef __FMB3DT_H_
#define __FMB3DT_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)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho,
         AABB3DTime* 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))
```

```
// Return x if x is negative, 0.0 else
#define neg(x) (x < 0.0 ? x : 0.0)

#define FST_VAR 0
#define SND_VAR 1
#define THD_VAR 2
#define FOR_VAR 3

#define EPSILON 0.0000001

// ------------ 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)[4],
  const double* Y,
     const int nbRows,
     const int nbCols,
        double (*Mp)[4],
        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 GetBound3DTime(
     const int iVar,
  const double (*M)[4],
  const double* Y,
     const int nbRows,
   AABB3DTime* const bdgBox);

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

// 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)[4],
  const double* Y,
     const int nbRows,
     const int nbCols,
        double (*Mp)[4],
        double* Yp,
    int* const nbRemainRows) {
```

```
// Initialize the number of rows in the result system
int nbResRows = 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
  double fabsMIRowIVar = fabs(M[iRow][iVar]);

  // If the coefficient for the eliminated vairable is not null
  // in this row
  if (fabsMIRowIVar > EPSILON) {

    // Shortcuts
    int sgnMIRowIVar = sgn(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]) &&
          fabs(M[jRow][iVar]) > EPSILON) {

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

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

            // Update the sum of the negative coefficient
            sumNegCoeff += neg(Mp[nbResRows][jCol]);

            // Increment the number of columns in the new inequality
            ++jCol;

          }

        }

        // Update the right side of the inequality
        Yp[nbResRows] =
```

```
          YIRowDivideByFabsMIRowIVar +
          Y[jRow] / fabs(M[jRow][iVar]);

        // If the right side of the inequality if lower than the sum of
        // negative coefficients in the row
        // (Add epsilon for numerical imprecision)
        if (Yp[nbResRows] < sumNegCoeff - EPSILON) {

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

        }

        // Increment the nb of rows into the result system
        ++nbResRows;

      }

    }

  }

}

// 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(MiRow[iVar]) < EPSILON) {

    // Shortcut
    double* MpnbResRows = Mp[nbResRows];

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

      if (iCol != iVar) {

        MpnbResRows[jCol] = MiRow[iCol];

        ++jCol;

      }

    }

    Yp[nbResRows] = Y[iRow];

    // Increment the nb of rows into the result system
    ++nbResRows;
```

```
    }

  }

  // Memorize the number of rows in the result system
  *nbRemainRows = nbResRows;

  // If we reach here the system is not inconsistent
  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)[4],
  const double* Y,
     const int nbRows,
   AABB3DTime* 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 < -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)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho,
        AABB3DTime* const bdgBox) {

  // Get the projection of the Frame 'tho' in Frame 'that' coordinates
  // system
  Frame3DTime thoProj;
  Frame3DTimeImportFrame(that, tho, &thoProj);

  // Declare two variables to memorize the system to be solved M.X <= Y
  // (M arrangement is [iRow][iCol])
  double M[14][4];
  double Y[14];

  // Create the inequality system

  // -V_jT-sum_iC_j,iX_i<=O_j
  M[0][0] = -thoProj.comp[0][0];
  M[0][1] = -thoProj.comp[1][0];
  M[0][2] = -thoProj.comp[2][0];
  M[0][3] = -thoProj.speed[0];
  Y[0] = thoProj.orig[0];
  if (Y[0] < neg(M[0][0]) + neg(M[0][1]) + neg(M[0][2]) + neg(M[0][3]))
    return false;

  M[1][0] = -thoProj.comp[0][1];
  M[1][1] = -thoProj.comp[1][1];
  M[1][2] = -thoProj.comp[2][1];
  M[1][3] = -thoProj.speed[1];
  Y[1] = thoProj.orig[1];
  if (Y[1] < neg(M[1][0]) + neg(M[1][1]) + neg(M[1][2]) + neg(M[1][3]))
    return false;
```

```
M[2][0] = -thoProj.comp[0][2];
M[2][1] = -thoProj.comp[1][2];
M[2][2] = -thoProj.comp[2][2];
M[2][3] = -thoProj.speed[2];
Y[2] = thoProj.orig[2];
if (Y[2] < neg(M[2][0]) + neg(M[2][1]) + neg(M[2][2]) + neg(M[2][3]))
  return false;

// Variable to memorise the nb of rows in the system
int nbRows = 3;

if (that->type == FrameCuboid) {

  // V_jT+sum_iC_j,iX_i<=1.0-O_j
  M[nbRows][0] = thoProj.comp[0][0];
  M[nbRows][1] = thoProj.comp[1][0];
  M[nbRows][2] = thoProj.comp[2][0];
  M[nbRows][3] = thoProj.speed[0];
  Y[nbRows] = 1.0 - thoProj.orig[0];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]) + neg(M[nbRows][3]))
    return false;
  ++nbRows;

  M[nbRows][0] = thoProj.comp[0][1];
  M[nbRows][1] = thoProj.comp[1][1];
  M[nbRows][2] = thoProj.comp[2][1];
  M[nbRows][3] = thoProj.speed[1];
  Y[nbRows] = 1.0 - thoProj.orig[1];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]) + neg(M[nbRows][3]))
    return false;
  ++nbRows;

  M[nbRows][0] = thoProj.comp[0][2];
  M[nbRows][1] = thoProj.comp[1][2];
  M[nbRows][2] = thoProj.comp[2][2];
  M[nbRows][3] = thoProj.speed[2];
  Y[nbRows] = 1.0 - thoProj.orig[2];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]) + neg(M[nbRows][3]))
    return false;
  ++nbRows;

} else {

  // sum_j(V_jT+sum_iC_j,iX_i)<=1.0-sum_iO_i
  M[nbRows][0] =
    thoProj.comp[0][0] + thoProj.comp[0][1] + thoProj.comp[0][2];
  M[nbRows][1] =
    thoProj.comp[1][0] + thoProj.comp[1][1] + thoProj.comp[1][2];
  M[nbRows][2] =
    thoProj.comp[2][0] + thoProj.comp[2][1] + thoProj.comp[2][2];
  M[nbRows][3] = thoProj.speed[0] + thoProj.speed[1] + thoProj.speed[2];
  Y[nbRows] = 1.0 - thoProj.orig[0] - thoProj.orig[1] - thoProj.orig[2];
  if (Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                  neg(M[nbRows][2]) + neg(M[nbRows][3]))
    return false;
  ++nbRows;

}
```

```
if (tho->type == FrameCuboid) {

  // X_i <= 1.0
  M[nbRows][0] = 1.0;
  M[nbRows][1] = 0.0;
  M[nbRows][2] = 0.0;
  M[nbRows][3] = 0.0;
  Y[nbRows] = 1.0;
  ++nbRows;

  M[nbRows][0] = 0.0;
  M[nbRows][1] = 1.0;
  M[nbRows][2] = 0.0;
  M[nbRows][3] = 0.0;
  Y[nbRows] = 1.0;
  ++nbRows;

  M[nbRows][0] = 0.0;
  M[nbRows][1] = 0.0;
  M[nbRows][2] = 1.0;
  M[nbRows][3] = 0.0;
  Y[nbRows] = 1.0;
  ++nbRows;

} else {

  // sum_iX_i <=1.0
  M[nbRows][0] = 1.0;
  M[nbRows][1] = 1.0;
  M[nbRows][2] = 1.0;
  M[nbRows][3] = 0.0;
  Y[nbRows] = 1.0;
  ++nbRows;

}

// -X_i <= 0.0
M[nbRows][0] = -1.0;
M[nbRows][1] = 0.0;
M[nbRows][2] = 0.0;
M[nbRows][3] = 0.0;
Y[nbRows] = 0.0;
++nbRows;

M[nbRows][0] = 0.0;
M[nbRows][1] = -1.0;
M[nbRows][2] = 0.0;
M[nbRows][3] = 0.0;
Y[nbRows] = 0.0;
++nbRows;

M[nbRows][0] = 0.0;
M[nbRows][1] = 0.0;
M[nbRows][2] = -1.0;
M[nbRows][3] = 0.0;
Y[nbRows] = 0.0;
++nbRows;

// 0.0 <= t <= 1.0
M[nbRows][0] = 0.0;
M[nbRows][1] = 0.0;
M[nbRows][2] = 0.0;
```

```
M[nbRows][3] = 1.0;
Y[nbRows] = 1.0;
++nbRows;

M[nbRows][0] = 0.0;
M[nbRows][1] = 0.0;
M[nbRows][2] = 0.0;
M[nbRows][3] = -1.0;
Y[nbRows] = 0.0;
++nbRows;

// Solve the system

// Declare a AABB to memorize the bounding box of the intersection
// in the coordinates system of that
AABB3DTime bdgBoxLocal;

// Declare variables to eliminate the first variable
// The size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mp[63][4];
//double Yp[63];
double Mp[22][4];
double Yp[22];
int nbRowsP;

// Eliminate the first variable in the original system
bool inconsistency =
  ElimVar3DTime(
    FST_VAR,
    M,
    Y,
    nbRows,
    4,
    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 size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mpp[1056][4];
//double Ypp[1056];
double Mpp[57][4];
double Ypp[57];
int nbRowsPP;

// Eliminate the second variable (which is the first in the new system)
inconsistency =
  ElimVar3DTime(
```

```
    FST_VAR,
    Mp,
    Yp,
    nbRowsP,
    3,
    Mpp,
    Ypp,
    &nbRowsPP);

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

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

}

// Declare variables to eliminate the third variable
// The size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mppp[279840][4];
//double Yppp[279840];
double Mppp[560][4];
double Yppp[560];
int nbRowsPPP;

// Eliminate the third variable (which is the first in the new system)
inconsistency =
  ElimVar3DTime(
    FST_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    2,
    Mppp,
    Yppp,
    &nbRowsPPP);

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

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

}

// Get the bounds for the remaining fourth variable
GetBound3DTime(
  FOR_VAR,
  Mppp,
  Yppp,
  nbRowsPPP,
  &bdgBoxLocal);

// If the bounds are inconstent
if (bdgBoxLocal.min[FOR_VAR] >= bdgBoxLocal.max[FOR_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;

}

// Eliminate the fourth variable (which is the second in the new
// system)
inconsistency =
  ElimVar3DTime(
    SND_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    2,
    Mppp,
    Yppp,
    &nbRowsPPP);

// Get the bounds for the remaining third variable
GetBound3DTime(
  THD_VAR,
  Mppp,
  Yppp,
  nbRowsPPP,
  &bdgBoxLocal);

// Now starts again from the initial systems and eliminate the
// third and fourth variables to get the bounds of the first and
// second variables.
// No need to check for consistency because we already know here
// that the Frames are intersecting and the system is consistent
inconsistency =
  ElimVar3DTime(
    FOR_VAR,
    M,
    Y,
    nbRows,
    4,
    Mp,
    Yp,
    &nbRowsP);

inconsistency =
  ElimVar3DTime(
    THD_VAR,
    Mp,
    Yp,
    nbRowsP,
    3,
    Mpp,
    Ypp,
    &nbRowsPP);

inconsistency =
  ElimVar3DTime(
    SND_VAR,
    Mpp,
```

```
        Ypp,
        nbRowsPP,
        2,
        Mppp,
        Yppp,
        &nbRowsPPP);

  GetBound3DTime(
    FST_VAR,
    Mppp,
    Yppp,
    nbRowsPPP,
    &bdgBoxLocal);

  inconsistency =
    ElimVar3DTime(
      FST_VAR,
      Mpp,
      Ypp,
      nbRowsPP,
      2,
      Mppp,
      Yppp,
      &nbRowsPPP);

  GetBound3DTime(
    SND_VAR,
    Mppp,
    Yppp,
    nbRowsPPP,
    &bdgBoxLocal);

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

    // Memorize the result
    *bdgBox = bdgBoxLocal;

  }

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

}
```

# 5 Minimal example of use

In this section I give a minimal example of how to use the code given in the previous section.

## 5.1 2D static

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

```c
// Include FMB algorithm library
#include "fmb2d.h"

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

  // 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(
      FrameCuboid,
      origP2D,
      compP2D);

  double origQ2D[2] = {0.0, 0.0};
  double compQ2D[2][2] = {
    {1.0, 0.0},
    {0.0, 1.0}};
  Frame2D Q2D =
    Frame2DCreateStatic(
      FrameCuboid,
      origQ2D,
      compQ2D);

  // Declare a variable to memorize the result of the intersection
  // detection
  AABB2D bdgBox2DLocal;

  // Test for intersection between P and Q
  bool isIntersecting2D =
    FMBTestIntersection2D(
      &P2D,
      &Q2D,
      &bdgBox2DLocal);

  // If the two objects are intersecting
  if (isIntersecting2D) {

    printf("Intersection detected in AABB ");

    // Export the local bounding box toward the real coordinates
    // system
    AABB2D bdgBox2D;
    Frame2DExportBdgBox(
      &Q2D,
      &bdgBox2DLocal,
      &bdgBox2D);

    // Clip with the AABB of 'Q2D' and 'P2D' to improve results
    for (int iAxis = 2;
         iAxis--;) {

      if (bdgBox2D.min[iAxis] < P2D.bdgBox.min[iAxis]) {

        bdgBox2D.min[iAxis] = P2D.bdgBox.min[iAxis];

      }
      if (bdgBox2D.max[iAxis] > P2D.bdgBox.max[iAxis]) {
```

```
        bdgBox2D.max[iAxis] = P2D.bdgBox.max[iAxis];

      }

      if (bdgBox2D.min[iAxis] < Q2D.bdgBox.min[iAxis]) {

        bdgBox2D.min[iAxis] = Q2D.bdgBox.min[iAxis];

      }
      if (bdgBox2D.max[iAxis] > Q2D.bdgBox.max[iAxis]) {

        bdgBox2D.max[iAxis] = Q2D.bdgBox.max[iAxis];

      }

    }

    AABB2DPrint(&bdgBox2D);
    printf("\n");

  // Else, the two objects are not intersecting
  } else {

    printf("No intersection.\n");

  }

  return 0;

}
```

## 5.2 3D static

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

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

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

  // 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] = {0.0, 0.0, 0.0};
  double compQ3D[3][3] = {
    {1.0, 0.0, 0.0},
    {0.0, 1.0, 0.0},
```

```
  {0.0, 0.0, 1.0}};
Frame3D Q3D =
  Frame3DCreateStatic(
    FrameTetrahedron,
    origQ3D,
    compQ3D);

// Declare a variable to memorize the result of the intersection
// detection
AABB3D bdgBox3DLocal;

// Test for intersection between P and Q
bool isIntersecting3D =
  FMBTestIntersection3D(
    &P3D,
    &Q3D,
    &bdgBox3DLocal);

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

  printf("Intersection detected in AABB ");

  // Export the local bounding box toward the real coordinates
  // system
  AABB3D bdgBox3D;
  Frame3DExportBdgBox(
    &Q3D,
    &bdgBox3DLocal,
    &bdgBox3D);

  // Clip with the AABB of 'Q3D' and 'P3D' to improve results
  for (int iAxis = 2;
       iAxis--;) {

    if (bdgBox3D.min[iAxis] < P3D.bdgBox.min[iAxis]) {

      bdgBox3D.min[iAxis] = P3D.bdgBox.min[iAxis];

    }
    if (bdgBox3D.max[iAxis] > P3D.bdgBox.max[iAxis]) {

      bdgBox3D.max[iAxis] = P3D.bdgBox.max[iAxis];

    }

    if (bdgBox3D.min[iAxis] < Q3D.bdgBox.min[iAxis]) {

      bdgBox3D.min[iAxis] = Q3D.bdgBox.min[iAxis];

    }
    if (bdgBox3D.max[iAxis] > Q3D.bdgBox.max[iAxis]) {

      bdgBox3D.max[iAxis] = Q3D.bdgBox.max[iAxis];

    }

  }

  AABB3DPrint(&bdgBox3D);
  printf("\n");
```

```
  // Else, the two objects are not intersecting
  } else {

    printf("No intersection.\n");

  }

  return 0;
}
```

## 5.3 2D dynamic

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

// Include FMB algorithm library
#include "fmb2dt.h"

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

  // Create the two objects to be tested for intersection
  double origP2DTime[2] = {0.0, 0.0};
  double speedP2DTime[2] = {0.0, 0.0};
  double compP2DTime[2][2] = {
    {1.0, 0.0},  // First component
    {0.0, 1.0}}; // Second component
  Frame2DTime P2DTime =
    Frame2DTimeCreateStatic(
      FrameCuboid,
      origP2DTime,
      speedP2DTime,
      compP2DTime);

  double origQ2DTime[2] = {0.0,0.0};
  double speedQ2DTime[2] = {0.0,0.0};
  double compQ2DTime[2][2] = {
    {1.0, 0.0},
    {0.0, 1.0}};
  Frame2DTime Q2DTime =
    Frame2DTimeCreateStatic(
      FrameCuboid,
      origQ2DTime,
      speedQ2DTime,
      compQ2DTime);

  // Declare a variable to memorize the result of the intersection
  // detection
  AABB2DTime bdgBox2DTimeLocal;

  // Test for intersection between P and Q
  bool isIntersecting2DTime =
    FMBTestIntersection2DTime(
      &P2DTime,
      &Q2DTime,
      &bdgBox2DTimeLocal);

  // If the two objects are intersecting
  if (isIntersecting2DTime) {
```

```
    printf("Intersection detected in AABB ");

    // Export the local bounding box toward the real coordinates
    // system
    AABB2DTime bdgBox2DTime;
    Frame2DTimeExportBdgBox(
      &Q2DTime,
      &bdgBox2DTimeLocal,
      &bdgBox2DTime);

    AABB2DTimePrint(&bdgBox2DTime);
    printf("\n");

  // Else, the two objects are not intersecting
  } else {

    printf("No intersection.\n");

  }

  return 0;
}
```

## 5.4 3D dynamic

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

// Include FMB algorithm library
#include "fmb3dt.h"

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

  // Create the two objects to be tested for intersection
  double origP3DTime[3] = {0.0, 0.0, 0.0};
  double speedP3DTime[3] = {0.0, 0.0, 0.0};
  double compP3DTime[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
  Frame3DTime P3DTime =
    Frame3DTimeCreateStatic(
      FrameCuboid,
      origP3DTime,
      speedP3DTime,
      compP3DTime);

  double origQ3DTime[3] = {0.0, 0.0, 0.0};
  double speedQ3DTime[3] = {0.0, 0.0, 0.0};
  double compQ3DTime[3][3] = {
    {1.0, 0.0, 0.0},
    {0.0, 1.0, 0.0},
    {0.0, 0.0, 1.0}};
  Frame3DTime Q3DTime =
    Frame3DTimeCreateStatic(
      FrameCuboid,
      origQ3DTime,
```

```
      speedQ3DTime,
      compQ3DTime);

  // Declare a variable to memorize the result of the intersection
  // detection
  AABB3DTime bdgBox3DTimeLocal;

  // Test for intersection between P and Q
  bool isIntersecting3DTime =
    FMBTestIntersection3DTime(
      &P3DTime,
      &Q3DTime,
      &bdgBox3DTimeLocal);

  // If the two objects are intersecting
  if (isIntersecting3DTime) {

    printf("Intersection detected in AABB ");

    // Export the local bounding box toward the real coordinates
    // system
    AABB3DTime bdgBox3DTime;
    Frame3DTimeExportBdgBox(
      &Q3DTime,
      &bdgBox3DTimeLocal,
      &bdgBox3DTime);

    AABB3DTimePrint(&bdgBox3DTime);
    printf("\n");

  // Else, the two objects are not intersecting
  } else {

    printf("No intersection.\n");

  }

  return 0;
}
```

# 6 Unit tests

In this section I introduce the code I've used to test the algorithm and its implementation. The test consists of running the algorithm on a set of cases for which the solution has been computed by hand. The code of the implementation of the SAT algorithm is given in annex (p.216)

## 6.1 Code

### 6.1.1 2D static

```
// 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 "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// 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;

// 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;

  }

}

// Main function
void Validate2D(void) {

  // Initialise the random generator
  srandom(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);

}

int main(int argc, char** argv) {

  printf("===== 2D static ======\n");
  Validate2D();

  return 0;
}
```

### 6.1.2 3D static

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

// Include FMB and SAT algorithm library
#include "fmb3d.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// 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[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 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;

  }

}

void Validate3D(void) {

  // Initialise the random generator
  srandom(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("===== 3D static ======\n");
  Validate3D();

  return 0;
}
```

### 6.1.3 2D dynamic

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

// Include FMB and SAT algorithm library
#include "fmb2dt.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// 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];
  double speed[2];
} Param2DTime;

// 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 Validation2DTime(
  const Param2DTime paramP,
  const Param2DTime paramQ) {

  // Create the two Frames
  Frame2DTime P =
    Frame2DTimeCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.speed,
      paramP.comp);

  Frame2DTime Q =
    Frame2DTimeCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.speed,
      paramQ.comp);

  // Helper variables to loop on the pair (that, tho) and (tho, that)
  Frame2DTime* that = &P;
  Frame2DTime* tho = &Q;

  // Loop on pairs of Frames
  for (int iPair = 2;
       iPair--;) {

    // Test intersection with FMB
    bool isIntersectingFMB =
      FMBTestIntersection2DTime(
        that,
        tho,
        NULL);

    // Test intersection with SAT
    bool isIntersectingSAT =
      SATTestIntersection2DTime(
        that,
        tho);

    // If the results are different
    if (isIntersectingFMB != isIntersectingSAT) {
```

```
      // Print the disagreement
      printf("Validation2D has failed\n");
      Frame2DTimePrint(that);
      printf(" against ");
      Frame2DTimePrint(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 Validate2DTime(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Declare two variables to memorize the arguments to the
  // Validation function
  Param2DTime paramP;
  Param2DTime 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
```

```
    Param2DTime* 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;
        param->speed[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
      Validation2DTime(
        paramP,
        paramQ);

    }

  }

  // If we reached it means the validation was successfull
  // Print results
  printf("Validation2DTime has succeed.\n");
  printf("Tested %lu intersections ", nbInter);
  printf("and %lu no intersections\n", nbNoInter);

}

int main(int argc, char** argv) {

  printf("===== 2D dynamic ======\n");
```

```
  Validate2DTime();

  return 0;
}
```

### 6.1.4 3D dynamic

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

// Include FMB and SAT algorithm library
#include "fmb3dt.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// 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[3];
  double comp[3][3];
  double speed[3];
} Param3DTime;

// 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 Validation3DTime(
  const Param3DTime paramP,
  const Param3DTime paramQ) {

  // Create the two Frames
  Frame3DTime P =
    Frame3DTimeCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.speed,
      paramP.comp);

  Frame3DTime Q =
    Frame3DTimeCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.speed,
      paramQ.comp);
```

```
// Helper variables to loop on the pair (that, tho) and (tho, that)
Frame3DTime* that = &P;
Frame3DTime* tho = &Q;

// Loop on pairs of Frames
for (int iPair = 2;
     iPair--;) {

  // Test intersection with FMB
  bool isIntersectingFMB =
    FMBTestIntersection3DTime(
      that,
      tho,
      NULL);

  // Test intersection with SAT
  bool isIntersectingSAT =
    SATTestIntersection3DTime(
      that,
      tho);

  // If the results are different
  if (isIntersectingFMB != isIntersectingSAT) {

    // Print the disagreement
    printf("Validation3D has failed\n");
    Frame3DTimePrint(that);
    printf(" against ");
    Frame3DTimePrint(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 Validate3DTime(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Declare two variables to memorize the arguments to the
  // Validation function
  Param3DTime paramP;
  Param3DTime 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
    Param3DTime* 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;
        param->speed[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
      Validation3DTime(
        paramP,
        paramQ);

    }

  }

  // If we reached it means the validation was successfull
  // Print results
  printf("Validation3DTime has succeed.\n");
  printf("Tested %lu intersections ", nbInter);
  printf("and %lu no intersections\n", nbNoInter);

}

int main(int argc, char** argv) {

  printf("===== 3D dynamic ======\n");
  Validate3DTime();

  return 0;
}
```

## 6.2 Results

### 6.2.1 2D static

```
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.500000,0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(0.500000,0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed
```

```
Co(-0.500000,-0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.500000,0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
 Succeed

Co(0.500000,0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(-0.500000,-0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
 Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.250000,-0.250000) x(0.500000,0.000000) y(0.000000,2.000000)
Succeed

Co(0.250000,-0.250000) x(0.500000,0.000000) y(0.000000,2.000000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(-0.250000,0.250000) x(2.000000,0.000000) y(0.000000,0.500000)
Succeed

Co(-0.250000,0.250000) x(2.000000,0.000000) y(0.000000,0.500000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,1.000000) y(-1.000000,1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.000000,0.000000) x(1.000000,1.000000) y(-1.000000,1.000000)
Succeed

Co(-0.500000,-0.500000) x(1.000000,1.000000) y(-1.000000,1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(-0.500000,-0.500000) x(1.000000,1.000000) y(-1.000000,1.000000)
Succeed

Co(1.500000,1.500000) x(1.000000,-1.000000) y(-1.000000,-1.000000)
against
Co(1.000000,0.000000) x(-1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(1.000000,0.000000) x(-1.000000,0.000000) y(0.000000,1.000000)
against
Co(1.500000,1.500000) x(1.000000,-1.000000) y(-1.000000,-1.000000)
Succeed

Co(1.000000,0.500000) x(-0.500000,0.500000) y(-0.500000,-0.500000)
```

```
against
Co(0.000000,1.000000) x(1.000000,0.000000) y(0.000000,-1.000000)
Succeed

Co(0.000000,1.000000) x(1.000000,0.000000) y(0.000000,-1.000000)
against
Co(1.000000,0.500000) x(-0.500000,0.500000) y(-0.500000,-0.500000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(1.000000,1.000000)
against
Co(2.000000,-1.000000) x(0.000000,1.000000) y(-0.500000,1.000000)
Succeed

Co(2.000000,-1.000000) x(0.000000,1.000000) y(-0.500000,1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(1.000000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
against
Co(1.000000,1.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
Succeed

Co(1.000000,1.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
against
Co(1.000000,2.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
Succeed

Co(1.000000,2.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
Succeed

To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
against
Co(1.000000,2.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
Succeed

Co(1.000000,2.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
against
To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
against
To(1.000000,2.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
Succeed

To(1.000000,2.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
```

```
Succeed

To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
To(0.000000,-0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

To(0.000000,-0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

Co(0.500000,0.500000) x(-0.500000,0.000000) y(0.000000,-0.500000)
against
To(0.000000,-0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

To(0.000000,-0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.500000,0.500000) x(-0.500000,0.000000) y(0.000000,-0.500000)
Succeed

Co(0.500000,0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
against
To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
 Succeed

To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
Co(0.500000,0.500000) x(1.000000,0.000000) y(0.000000,1.000000)
 Succeed

Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
To(1.500000,1.500000) x(-1.500000,0.000000) y(0.000000,-1.500000)
Succeed

To(1.500000,1.500000) x(-1.500000,0.000000) y(0.000000,-1.500000)
against
Co(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed

To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
To(1.010000,1.010000) x(-1.000000,0.000000) y(0.000000,-1.000000)
 Succeed

To(1.010000,1.010000) x(-1.000000,0.000000) y(0.000000,-1.000000)
against
To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
 Succeed

To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
against
To(1.000000,1.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
Succeed
```

```
To(1.000000,1.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
against
To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
Succeed

To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
against
To(1.010000,1.500000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
 Succeed

To(1.010000,1.500000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
against
To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
 Succeed

All unit tests 2D have succeed.
```

### 6.2.2 3D static

```
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed

Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed

Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(0.500000,0.500000,0.500000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed

Co(0.500000,0.500000,0.500000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed

Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(-0.500000,-0.500000,-0.500000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed

Co(-0.500000,-0.500000,-0.500000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed
```

```
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(1.500000,1.500000,1.500000) x(-1.000000,0.000000,0.000000) y
    (0.000000,-1.000000,0.000000) z(0.000000,0.000000,-1.000000)
Succeed

Co(1.500000,1.500000,1.500000) x(-1.000000,0.000000,0.000000) y
    (0.000000,-1.000000,0.000000) z(0.000000,0.000000,-1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed

Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(0.500000,1.500000,-1.500000) x(1.000000,0.000000,0.000000) y
    (0.000000,-1.000000,0.000000) z(0.000000,0.000000,1.000000)
 Succeed

Co(0.500000,1.500000,-1.500000) x(1.000000,0.000000,0.000000) y
    (0.000000,-1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
 Succeed

Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,-1.000000)
against
Co(0.500000,1.500000,-1.500000) x(1.000000,0.000000,0.000000) y
    (0.000000,-1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed

Co(0.500000,1.500000,-1.500000) x(1.000000,0.000000,0.000000) y
    (0.000000,-1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,-1.000000)
Succeed

Co(-1.010000,-1.010000,-1.010000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
 Succeed

Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(-1.010000,-1.010000,-1.010000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
 Succeed

Co(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed
```

```
Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
Succeed

To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
 Succeed

Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
 Succeed

Co(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed

To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
Succeed

To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
 Succeed

To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
 Succeed

To(-0.500000,-1.000000,-0.500000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed

To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
To(-0.500000,-1.000000,-0.500000) x(1.000000,0.000000,0.000000) y
    (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
```

```
Succeed

All unit tests 3D have succeed.
```

### 6.2.3 2D dynamic

```
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
against
Co(-1.000000,0.000000) s(-1.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
 Succeed

Co(-1.000000,0.000000) s(-1.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
 Succeed

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
against
Co(-1.010000,-1.010000) s(1.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
 Succeed

Co(-1.010000,-1.010000) s(1.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
 Succeed

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
against
Co(-1.000000,0.000000) s(1.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
Succeed

Co(-1.000000,0.000000) s(1.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
Succeed

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
against
Co(-1.000000,0.250000) s(4.000000,0.000000) x(0.500000,0.000000) y
    (0.000000,0.500000)
Succeed

Co(-1.000000,0.250000) s(4.000000,0.000000) x(0.500000,0.000000) y
    (0.000000,0.500000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
Succeed
```

```
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
against
Co(0.250000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
    (0.000000,0.500000)
Succeed

Co(0.250000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
    (0.000000,0.500000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
Succeed

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
against
Co(0.900000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
    (0.000000,0.500000)
Succeed

Co(0.900000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
    (0.000000,0.500000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
Succeed

All unit tests 2DTime have succeed.
```

### 6.2.4 3D dynamic

```
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(-1.000000,0.000000,0.000000) s(-1.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
 Succeed

Co(-1.000000,0.000000,0.000000) s(-1.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
 Succeed

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(-1.010000,-1.010000,0.000000) s(1.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
 Succeed

Co(-1.010000,-1.010000,0.000000) s(1.000000,0.000000,0.000000) x
```

```
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
 Succeed

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(-1.000000,0.000000,0.000000) s(1.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
Succeed

Co(-1.000000,0.000000,0.000000) s(1.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
Succeed

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(-1.000000,0.250000,0.000000) s(4.000000,0.000000,0.000000) x
    (0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
    (0.000000,0.000000,1.000000)
Succeed

Co(-1.000000,0.250000,0.000000) s(4.000000,0.000000,0.000000) x
    (0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
Succeed

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(0.250000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
    (0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
    (0.000000,0.000000,1.000000)
Succeed

Co(0.250000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
    (0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
Succeed
```

```
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(0.900000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
    (0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
    (0.000000,0.000000,1.000000)
Succeed

Co(0.900000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
    (0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
    (0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
    (1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
    (0.000000,0.000000,1.000000)
Succeed

All unit tests 3DTime have succeed.
```

# 7 Validation against SAT

In this section I introduce the code I've used to validate the algorithm and its implementation. The validation consists of 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.216)

## 7.1 Code

### 7.1.1 2D static

```
// 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 "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// 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;

// 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;

  }

}

// Main function
void Validate2D(void) {

  // Initialise the random generator
  srandom(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);

}

int main(int argc, char** argv) {

  printf("===== 2D static ======\n");
  Validate2D();

  return 0;
}
```

### 7.1.2 3D static

```
// Include standard libraries
```

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>

// Include FMB and SAT algorithm library
#include "fmb3d.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// 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[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 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;

  }

}

void Validate3D(void) {

  // Initialise the random generator
  srandom(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("===== 3D static ======\n");
  Validate3D();

  return 0;
}
```

### 7.1.3 2D dynamic

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

// Include FMB and SAT algorithm library
#include "fmb2dt.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// 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];
  double speed[2];
} Param2DTime;

// 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 Validation2DTime(
  const Param2DTime paramP,
  const Param2DTime paramQ) {

  // Create the two Frames
  Frame2DTime P =
    Frame2DTimeCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.speed,
      paramP.comp);

  Frame2DTime Q =
    Frame2DTimeCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.speed,
      paramQ.comp);

  // Helper variables to loop on the pair (that, tho) and (tho, that)
  Frame2DTime* that = &P;
  Frame2DTime* tho = &Q;

  // Loop on pairs of Frames
  for (int iPair = 2;
       iPair--;) {

    // Test intersection with FMB
    bool isIntersectingFMB =
      FMBTestIntersection2DTime(
        that,
        tho,
        NULL);

    // Test intersection with SAT
    bool isIntersectingSAT =
      SATTestIntersection2DTime(
        that,
        tho);

    // If the results are different
    if (isIntersectingFMB != isIntersectingSAT) {

      // Print the disagreement
      printf("Validation2D has failed\n");
      Frame2DTimePrint(that);
      printf(" against ");
      Frame2DTimePrint(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 Validate2DTime(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Declare two variables to memorize the arguments to the
  // Validation function
  Param2DTime paramP;
  Param2DTime 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
    Param2DTime* 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;
        param->speed[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
      Validation2DTime(
        paramP,
        paramQ);

    }

  }

  // If we reached it means the validation was successfull
  // Print results
  printf("Validation2DTime has succeed.\n");
  printf("Tested %lu intersections ", nbInter);
  printf("and %lu no intersections\n", nbNoInter);

}

int main(int argc, char** argv) {

  printf("===== 2D dynamic ======\n");
  Validate2DTime();

  return 0;
}
```

### 7.1.4 3D dynamic

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

// Include FMB and SAT algorithm library
#include "fmb3dt.h"
#include "sat.h"
```

```
// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// 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[3];
  double comp[3][3];
  double speed[3];
} Param3DTime;

// 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 Validation3DTime(
  const Param3DTime paramP,
  const Param3DTime paramQ) {

  // Create the two Frames
  Frame3DTime P =
    Frame3DTimeCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.speed,
      paramP.comp);

  Frame3DTime Q =
    Frame3DTimeCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.speed,
      paramQ.comp);

  // Helper variables to loop on the pair (that, tho) and (tho, that)
  Frame3DTime* that = &P;
  Frame3DTime* tho = &Q;

  // Loop on pairs of Frames
  for (int iPair = 2;
       iPair--;) {

    // Test intersection with FMB
    bool isIntersectingFMB =
      FMBTestIntersection3DTime(
        that,
        tho,
        NULL);

    // Test intersection with SAT
    bool isIntersectingSAT =
```

```
      SATTestIntersection3DTime(
        that,
        tho);

    // If the results are different
    if (isIntersectingFMB != isIntersectingSAT) {

      // Print the disagreement
      printf("Validation3D has failed\n");
      Frame3DTimePrint(that);
      printf(" against ");
      Frame3DTimePrint(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 Validate3DTime(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Declare two variables to memorize the arguments to the
  // Validation function
  Param3DTime paramP;
  Param3DTime 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
  Param3DTime* 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;
      param->speed[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
    Validation3DTime(
      paramP,
      paramQ);
```

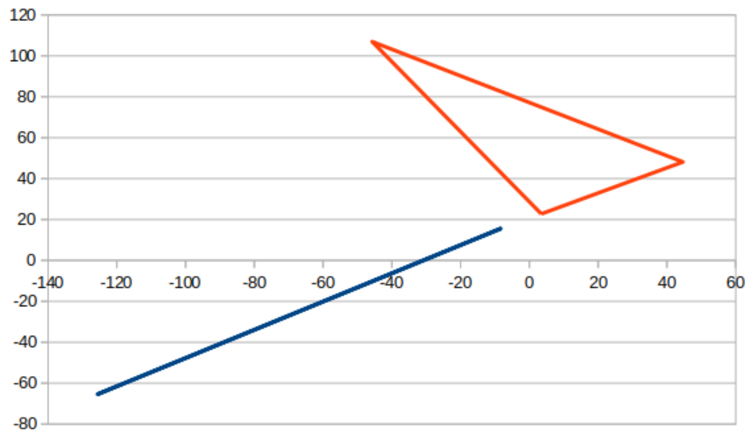
```
    }

  }

  // If we reached it means the validation was successfull
  // Print results
  printf("Validation3DTime has succeed.\n");
  printf("Tested %lu intersections ", nbInter);
  printf("and %lu no intersections\n", nbNoInter);

}

int main(int argc, char** argv) {

  printf("===== 3D dynamic ======\n");
  Validate3DTime();

  return 0;
}
```

## 7.2 Results

### 7.2.1 Failures

Validation has failed in one case: when one or both of the frame are degenerated (at least two of there components ae colinear). An example is given below for reference:

```
    ===== 2D static ======
Validation2D has failed
Co(-63.571705,-22.581119) x(55.239119,38.152177) y(-62.031537,-42.843548) against To(3.474294,22.751011)
x(-49.195251,84.166201) y(41.179031,-95.350316)
FMB : intersection
SAT : no intersection
```



This case can be detected and avoided prior to the intersection test by checking the determinant of the frame: degenerated frames have a null determinant. In the example above the determinant of the first frame is equal to -0.001667.

### 7.2.2 2D static

```
===== 2D static ======
Validation2D has succeed.
Tested 469100 intersections and 1530820 no intersections
```

### 7.2.3 2D dynamic

```
===== 2D dynamic ======
Validation2DTime has succeed.
Tested 743186 intersections and 1256734 no intersections
```

### 7.2.4 3D static

```
===== 3D static ======
Validation3D has succeed.
Tested 315990 intersections and 1684010 no intersections
```

### 7.2.5 3D dynamic

```
===== 3D dynamic ======
Validation3DTime has succeed.
Tested 522396 intersections and 1477604 no intersections
```

# 8 Qualification against SAT

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.

## 8.1 Code

### 8.1.1 2D static

```
// 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 "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 100000
// 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_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;

// 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;

double minInterCC;
double maxInterCC;
double sumInterCC;
unsigned long countInterCC;
double minNoInterCC;
double maxNoInterCC;
double sumNoInterCC;
unsigned long countNoInterCC;

double minInterCT;
double maxInterCT;
double sumInterCT;
unsigned long countInterCT;
double minNoInterCT;
double maxNoInterCT;
double sumNoInterCT;
unsigned long countNoInterCT;

double minInterTC;
double maxInterTC;
double sumInterTC;
unsigned long countInterTC;
double minNoInterTC;
double maxNoInterTC;
double sumNoInterTC;
```

```
unsigned long countNoInterTC;

double minInterTT;
double maxInterTT;
double sumInterTT;
unsigned long countInterTT;
double minNoInterTT;
double maxNoInterTT;
double sumNoInterTT;
unsigned long countNoInterTT;

// 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 Qualification2DStatic(
        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[0] == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT[0] == 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;

    if (paramP.type == FrameCuboid &&
        paramQ.type == FrameCuboid) {

      if (countInterCC == 0) {

        minInterCC = ratio;
        maxInterCC = ratio;

      } else {

        if (minInterCC > ratio)
          minInterCC = ratio;
        if (maxInterCC < ratio)
          maxInterCC = ratio;

      }
      sumInterCC += ratio;
      ++countInterCC;
```

```
  } else if (paramP.type == FrameCuboid &&
             paramQ.type == FrameTetrahedron) {

    if (countInterCT == 0) {

      minInterCT = ratio;
      maxInterCT = ratio;

    } else {

      if (minInterCT > ratio)
        minInterCT = ratio;
      if (maxInterCT < ratio)
        maxInterCT = ratio;

    }
    sumInterCT += ratio;
    ++countInterCT;

  } else if (paramP.type == FrameTetrahedron &&
             paramQ.type == FrameCuboid) {

    if (countInterTC == 0) {

      minInterTC = ratio;
      maxInterTC = ratio;

    } else {

      if (minInterTC > ratio)
        minInterTC = ratio;
      if (maxInterTC < ratio)
        maxInterTC = ratio;

    }
    sumInterTC += ratio;
    ++countInterTC;

  } else if (paramP.type == FrameTetrahedron &&
             paramQ.type == FrameTetrahedron) {

    if (countInterTT == 0) {

      minInterTT = ratio;
      maxInterTT = ratio;

    } else {

      if (minInterTT > ratio)
        minInterTT = ratio;
      if (maxInterTT < ratio)
        maxInterTT = ratio;

    }
    sumInterTT += ratio;
    ++countInterTT;

  }

// 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;

if (paramP.type == FrameCuboid &&
    paramQ.type == FrameCuboid) {

  if (countNoInterCC == 0) {

    minNoInterCC = ratio;
    maxNoInterCC = ratio;

  } else {

    if (minNoInterCC > ratio)
      minNoInterCC = ratio;
    if (maxNoInterCC < ratio)
      maxNoInterCC = ratio;

  }
  sumNoInterCC += ratio;
  ++countNoInterCC;

} else if (paramP.type == FrameCuboid &&
           paramQ.type == FrameTetrahedron) {

  if (countNoInterCT == 0) {

    minNoInterCT = ratio;
    maxNoInterCT = ratio;

  } else {

    if (minNoInterCT > ratio)
      minNoInterCT = ratio;
    if (maxNoInterCT < ratio)
      maxNoInterCT = ratio;

  }
  sumNoInterCT += ratio;
  ++countNoInterCT;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameCuboid) {

  if (countNoInterTC == 0) {

    minNoInterTC = ratio;
    maxNoInterTC = ratio;
```

```
          } else {

            if (minNoInterTC > ratio)
              minNoInterTC = ratio;
            if (maxNoInterTC < ratio)
              maxNoInterTC = ratio;

          }
          sumNoInterTC += ratio;
          ++countNoInterTC;

        } else if (paramP.type == FrameTetrahedron &&
                   paramQ.type == FrameTetrahedron) {

          if (countNoInterTT == 0) {

            minNoInterTT = ratio;
            maxNoInterTT = ratio;

          } else {

            if (minNoInterTT > ratio)
              minNoInterTT = ratio;
            if (maxNoInterTT < ratio)
              maxNoInterTT = ratio;

          }
          sumNoInterTT += ratio;
          ++countNoInterTT;

        }
      }

    // 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 Qualify2DStatic(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Loop on runs
```

```
for (int iRun = 0;
     iRun < NB_RUNS;
     ++iRun) {

  // Ratio intersection/no intersection for the displayed results
  double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

  // Initialize counters
  minInter = 0.0;
  maxInter = 0.0;
  sumInter = 0.0;
  countInter = 0;
  minNoInter = 0.0;
  maxNoInter = 0.0;
  sumNoInter = 0.0;
  countNoInter = 0;

  minInterCC = 0.0;
  maxInterCC = 0.0;
  sumInterCC = 0.0;
  countInterCC = 0;
  minNoInterCC = 0.0;
  maxNoInterCC = 0.0;
  sumNoInterCC = 0.0;
  countNoInterCC = 0;

  minInterCT = 0.0;
  maxInterCT = 0.0;
  sumInterCT = 0.0;
  countInterCT = 0;
  minNoInterCT = 0.0;
  maxNoInterCT = 0.0;
  sumNoInterCT = 0.0;
  countNoInterCT = 0;

  minInterTC = 0.0;
  maxInterTC = 0.0;
  sumInterTC = 0.0;
  countInterTC = 0;
  minNoInterTC = 0.0;
  maxNoInterTC = 0.0;
  sumNoInterTC = 0.0;
  countNoInterTC = 0;

  minInterTT = 0.0;
  maxInterTT = 0.0;
  sumInterTT = 0.0;
  countInterTT = 0;
  minNoInterTT = 0.0;
  maxNoInterTT = 0.0;
  sumNoInterTT = 0.0;
  countNoInterTT = 0;

  // Declare two variables to memozie 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
    Qualification2DStatic(
      paramP,
      paramQ);

  }

}

// Display the results
if (iRun == 0) {

  printf("percPairInter\t");
  printf("countInter\tcountNoInter\t");
  printf("minInter\tavgInter\tmaxInter\t");
  printf("minNoInter\tavgNoInter\tmaxNoInter\t");
  printf("minTotal\tavgTotal\tmaxTotal\t");

  printf("countInterCC\tcountNoInterCC\t");
```

```
    printf("minInterCC\tavgInterCC\tmaxInterCC\t");
    printf("minNoInterCC\tavgNoInterCC\tmaxNoInterCC\t");
    printf("minTotalCC\tavgTotalCC\tmaxTotalCC\t");

    printf("countInterCT\tcountNoInterCT\t");
    printf("minInterCT\tavgInterCT\tmaxInterCT\t");
    printf("minNoInterCT\tavgNoInterCT\tmaxNoInterCT\t");
    printf("minTotalCT\tavgTotalCT\tmaxTotalCT\t");

    printf("countInterTC\tcountNoInterTC\t");
    printf("minInterTC\tavgInterTC\tmaxInterTC\t");
    printf("minNoInterTC\tavgNoInterTC\tmaxNoInterTC\t");
    printf("minTotalTC\tavgTotalTC\tmaxTotalTC\t");

    printf("countInterTT\tcountNoInterTT\t");
    printf("minInterTT\tavgInterTT\tmaxInterTT\t");
    printf("minNoInterTT\tavgNoInterTT\tmaxNoInterTT\t");
    printf("minTotalTT\tavgTotalTT\tmaxTotalTT\n");

  }

  printf("%.1f\t", ratioInter);

  printf("%lu\t%lu\t", 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 =
    ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
  printf("%f\t%f\t%f\t",
    (minNoInter < minInter ? minNoInter : minInter),
    avg,
    (maxNoInter > maxInter ? maxNoInter : maxInter));

  printf("%lu\t%lu\t", countInterCC, countNoInterCC);
  double avgInterCC = sumInterCC / (double)countInterCC;
  printf("%f\t%f\t%f\t", minInterCC, avgInterCC, maxInterCC);
  double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
  printf("%f\t%f\t%f\t", minNoInterCC, avgNoInterCC, maxNoInterCC);
  double avgCC =
    ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
  printf("%f\t%f\t%f\t",
    (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
    avgCC,
    (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));

  printf("%lu\t%lu\t", countInterCT, countNoInterCT);
  double avgInterCT = sumInterCT / (double)countInterCT;
  printf("%f\t%f\t%f\t", minInterCT, avgInterCT, maxInterCT);
  double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
  printf("%f\t%f\t%f\t", minNoInterCT, avgNoInterCT, maxNoInterCT);
  double avgCT =
    ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
  printf("%f\t%f\t%f\t",
    (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
    avgCT,
    (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));

  printf("%lu\t%lu\t", countInterTC, countNoInterTC);
  double avgInterTC = sumInterTC / (double)countInterTC;
  printf("%f\t%f\t%f\t", minInterTC, avgInterTC, maxInterTC);
```

```
    double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
    printf("%f\t%f\t%f\t", minNoInterTC, avgNoInterTC, maxNoInterTC);
    double avgTC =
      ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
    printf("%f\t%f\t%f\t",
      (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
      avgTC,
      (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));

    printf("%lu\t%lu\t", countInterTT, countNoInterTT);
    double avgInterTT = sumInterTT / (double)countInterTT;
    printf("%f\t%f\t%f\t", minInterTT, avgInterTT, maxInterTT);
    double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
    printf("%f\t%f\t%f\t", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
      ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    printf("%f\t%f\t%f\n",
      (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
      avgTT,
      (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));

  }

}

int main(int argc, char** argv) {

  Qualify2DStatic();

  return 0;
}
```

### 8.1.2 3D static

```
// 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 "fmb3d.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 100000
// 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

// 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[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;

double minInterCC;
double maxInterCC;
double sumInterCC;
unsigned long countInterCC;
double minNoInterCC;
double maxNoInterCC;
double sumNoInterCC;
unsigned long countNoInterCC;

double minInterCT;
double maxInterCT;
double sumInterCT;
unsigned long countInterCT;
double minNoInterCT;
double maxNoInterCT;
double sumNoInterCT;
unsigned long countNoInterCT;

double minInterTC;
double maxInterTC;
double sumInterTC;
unsigned long countInterTC;
double minNoInterTC;
double maxNoInterTC;
double sumNoInterTC;
unsigned long countNoInterTC;

double minInterTT;
double maxInterTT;
double sumInterTT;
unsigned long countInterTT;
double minNoInterTT;
double maxNoInterTT;
double sumNoInterTT;
unsigned long countNoInterTT;

// 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 Qualification3DStatic(
        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[0] == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT[0] == 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;

  if (paramP.type == FrameCuboid &&
      paramQ.type == FrameCuboid) {

    if (countInterCC == 0) {

      minInterCC = ratio;
      maxInterCC = ratio;

    } else {

      if (minInterCC > ratio)
        minInterCC = ratio;
      if (maxInterCC < ratio)
        maxInterCC = ratio;

    }
    sumInterCC += ratio;
    ++countInterCC;

  } else if (paramP.type == FrameCuboid &&
             paramQ.type == FrameTetrahedron) {

    if (countInterCT == 0) {

      minInterCT = ratio;
      maxInterCT = ratio;

    } else {

      if (minInterCT > ratio)
        minInterCT = ratio;
      if (maxInterCT < ratio)
        maxInterCT = ratio;

    }
    sumInterCT += ratio;
    ++countInterCT;

  } else if (paramP.type == FrameTetrahedron &&
```

```
          paramQ.type == FrameCuboid) {

    if (countInterTC == 0) {

      minInterTC = ratio;
      maxInterTC = ratio;

    } else {

      if (minInterTC > ratio)
        minInterTC = ratio;
      if (maxInterTC < ratio)
        maxInterTC = ratio;

    }
    sumInterTC += ratio;
    ++countInterTC;

  } else if (paramP.type == FrameTetrahedron &&
          paramQ.type == FrameTetrahedron) {

    if (countInterTT == 0) {

      minInterTT = ratio;
      maxInterTT = ratio;

    } else {

      if (minInterTT > ratio)
        minInterTT = ratio;
      if (maxInterTT < ratio)
        maxInterTT = ratio;

    }
    sumInterTT += ratio;
    ++countInterTT;

  }

// 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;

  if (paramP.type == FrameCuboid &&
      paramQ.type == FrameCuboid) {
```

```
  if (countNoInterCC == 0) {

    minNoInterCC = ratio;
    maxNoInterCC = ratio;

  } else {

    if (minNoInterCC > ratio)
      minNoInterCC = ratio;
    if (maxNoInterCC < ratio)
      maxNoInterCC = ratio;

  }
  sumNoInterCC += ratio;
  ++countNoInterCC;

} else if (paramP.type == FrameCuboid &&
           paramQ.type == FrameTetrahedron) {

  if (countNoInterCT == 0) {

    minNoInterCT = ratio;
    maxNoInterCT = ratio;

  } else {

    if (minNoInterCT > ratio)
      minNoInterCT = ratio;
    if (maxNoInterCT < ratio)
      maxNoInterCT = ratio;

  }
  sumNoInterCT += ratio;
  ++countNoInterCT;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameCuboid) {

  if (countNoInterTC == 0) {

    minNoInterTC = ratio;
    maxNoInterTC = ratio;

  } else {

    if (minNoInterTC > ratio)
      minNoInterTC = ratio;
    if (maxNoInterTC < ratio)
      maxNoInterTC = ratio;

  }
  sumNoInterTC += ratio;
  ++countNoInterTC;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameTetrahedron) {

  if (countNoInterTT == 0) {

    minNoInterTT = ratio;
    maxNoInterTT = ratio;
```

```
        } else {

          if (minNoInterTT > ratio)
            minNoInterTT = ratio;
          if (maxNoInterTT < ratio)
            maxNoInterTT = ratio;

        }
        sumNoInterTT += ratio;
        ++countNoInterTT;

      }
    }

  // 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 Qualify3DStatic(void) {

 // Initialise the random generator
 srandom(time(NULL));

 // Loop on runs
 for (int iRun = 0;
      iRun < NB_RUNS;
      ++iRun) {

  // Ratio intersection/no intersection for the displayed results
  double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

  // Initialize counters
  minInter = 0.0;
  maxInter = 0.0;
  sumInter = 0.0;
  countInter = 0;
  minNoInter = 0.0;
  maxNoInter = 0.0;
  sumNoInter = 0.0;
  countNoInter = 0;

  minInterCC = 0.0;
  maxInterCC = 0.0;
  sumInterCC = 0.0;
```

```
countInterCC = 0;
minNoInterCC = 0.0;
maxNoInterCC = 0.0;
sumNoInterCC = 0.0;
countNoInterCC = 0;

minInterCT = 0.0;
maxInterCT = 0.0;
sumInterCT = 0.0;
countInterCT = 0;
minNoInterCT = 0.0;
maxNoInterCT = 0.0;
sumNoInterCT = 0.0;
countNoInterCT = 0;

minInterTC = 0.0;
maxInterTC = 0.0;
sumInterTC = 0.0;
countInterTC = 0;
minNoInterTC = 0.0;
maxNoInterTC = 0.0;
sumNoInterTC = 0.0;
countNoInterTC = 0;

minInterTT = 0.0;
maxInterTT = 0.0;
sumInterTT = 0.0;
countInterTT = 0;
minNoInterTT = 0.0;
maxNoInterTT = 0.0;
sumNoInterTT = 0.0;
countNoInterTT = 0;

// Declare two variables to memozie 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
      Qualification3DStatic(
        paramP,
        paramQ);

    }

  }

  // Display the results
  if (iRun == 0) {

    printf("percPairInter\t");
    printf("countInter\tcountNoInter\t");
    printf("minInter\tavgInter\tmaxInter\t");
    printf("minNoInter\tavgNoInter\tmaxNoInter\t");
    printf("minTotal\tavgTotal\tmaxTotal\t");

    printf("countInterCC\tcountNoInterCC\t");
    printf("minInterCC\tavgInterCC\tmaxInterCC\t");
    printf("minNoInterCC\tavgNoInterCC\tmaxNoInterCC\t");
    printf("minTotalCC\tavgTotalCC\tmaxTotalCC\t");

    printf("countInterCT\tcountNoInterCT\t");
    printf("minInterCT\tavgInterCT\tmaxInterCT\t");
    printf("minNoInterCT\tavgNoInterCT\tmaxNoInterCT\t");
    printf("minTotalCT\tavgTotalCT\tmaxTotalCT\t");

    printf("countInterTC\tcountNoInterTC\t");
    printf("minInterTC\tavgInterTC\tmaxInterTC\t");
    printf("minNoInterTC\tavgNoInterTC\tmaxNoInterTC\t");
    printf("minTotalTC\tavgTotalTC\tmaxTotalTC\t");
```

```
  printf("countInterTT\tcountNoInterTT\t");
  printf("minInterTT\tavgInterTT\tmaxInterTT\t");
  printf("minNoInterTT\tavgNoInterTT\tmaxNoInterTT\t");
  printf("minTotalTT\tavgTotalTT\tmaxTotalTT\n");

}

printf("%.1f\t", ratioInter);

printf("%lu\t%lu\t", 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 =
  ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
printf("%f\t%f\t%f\t",
  (minNoInter < minInter ? minNoInter : minInter),
  avg,
  (maxNoInter > maxInter ? maxNoInter : maxInter));

printf("%lu\t%lu\t", countInterCC, countNoInterCC);
double avgInterCC = sumInterCC / (double)countInterCC;
printf("%f\t%f\t%f\t", minInterCC, avgInterCC, maxInterCC);
double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
printf("%f\t%f\t%f\t", minNoInterCC, avgNoInterCC, maxNoInterCC);
double avgCC =
  ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
printf("%f\t%f\t%f\t",
  (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
  avgCC,
  (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));

printf("%lu\t%lu\t", countInterCT, countNoInterCT);
double avgInterCT = sumInterCT / (double)countInterCT;
printf("%f\t%f\t%f\t", minInterCT, avgInterCT, maxInterCT);
double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
printf("%f\t%f\t%f\t", minNoInterCT, avgNoInterCT, maxNoInterCT);
double avgCT =
  ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
printf("%f\t%f\t%f\t",
  (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
  avgCT,
  (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));

printf("%lu\t%lu\t", countInterTC, countNoInterTC);
double avgInterTC = sumInterTC / (double)countInterTC;
printf("%f\t%f\t%f\t", minInterTC, avgInterTC, maxInterTC);
double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
printf("%f\t%f\t%f\t", minNoInterTC, avgNoInterTC, maxNoInterTC);
double avgTC =
  ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
printf("%f\t%f\t%f\t",
  (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
  avgTC,
  (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));

printf("%lu\t%lu\t", countInterTT, countNoInterTT);
double avgInterTT = sumInterTT / (double)countInterTT;
printf("%f\t%f\t%f\t", minInterTT, avgInterTT, maxInterTT);
double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
```

```
    printf("%f\t%f\t%f\t", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
      ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    printf("%f\t%f\t%f\n",
      (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
      avgTT,
      (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));

  }

}

int main(int argc, char** argv) {

  Qualify3DStatic();

  return 0;
}
```

### 8.1.3 2D dynamic

```
// 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 "fmb2dt.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 100000
// 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_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];
  double speed[2];
} Param2DTime;

// 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;

double minInterCC;
double maxInterCC;
double sumInterCC;
unsigned long countInterCC;
double minNoInterCC;
double maxNoInterCC;
double sumNoInterCC;
unsigned long countNoInterCC;

double minInterCT;
double maxInterCT;
double sumInterCT;
unsigned long countInterCT;
double minNoInterCT;
double maxNoInterCT;
double sumNoInterCT;
unsigned long countNoInterCT;

double minInterTC;
double maxInterTC;
double sumInterTC;
unsigned long countInterTC;
double minNoInterTC;
double maxNoInterTC;
double sumNoInterTC;
unsigned long countNoInterTC;

double minInterTT;
double maxInterTT;
double sumInterTT;
unsigned long countInterTT;
double minNoInterTT;
double maxNoInterTT;
double sumNoInterTT;
unsigned long countNoInterTT;

// 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 Qualification2DDynamic(
        const Param2DTime paramP,
        const Param2DTime paramQ) {

  // Create the two Frames
  Frame2DTime P =
    Frame2DTimeCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.speed,
      paramP.comp);

  Frame2DTime Q =
    Frame2DTimeCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.speed,
      paramQ.comp);
```

```
// Helper variables to loop on the pair (that, tho) and (tho, that)
Frame2DTime* that = &P;
Frame2DTime* 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] =
      FMBTestIntersection2DTime(
        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] =
    SATTestIntersection2DTime(
      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");
    Frame2DTimePrint(that);
    printf(" against ");
    Frame2DTimePrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB[0] == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT[0] == 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;

if (paramP.type == FrameCuboid &&
    paramQ.type == FrameCuboid) {

  if (countInterCC == 0) {

    minInterCC = ratio;
    maxInterCC = ratio;

  } else {

    if (minInterCC > ratio)
      minInterCC = ratio;
    if (maxInterCC < ratio)
      maxInterCC = ratio;

  }
  sumInterCC += ratio;
  ++countInterCC;

} else if (paramP.type == FrameCuboid &&
           paramQ.type == FrameTetrahedron) {

  if (countInterCT == 0) {

    minInterCT = ratio;
    maxInterCT = ratio;

  } else {

    if (minInterCT > ratio)
      minInterCT = ratio;
    if (maxInterCT < ratio)
      maxInterCT = ratio;

  }
  sumInterCT += ratio;
  ++countInterCT;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameCuboid) {

  if (countInterTC == 0) {

    minInterTC = ratio;
    maxInterTC = ratio;

  } else {

    if (minInterTC > ratio)
```

```
        minInterTC = ratio;
      if (maxInterTC < ratio)
        maxInterTC = ratio;

    }
    sumInterTC += ratio;
    ++countInterTC;

  } else if (paramP.type == FrameTetrahedron &&
             paramQ.type == FrameTetrahedron) {

    if (countInterTT == 0) {

      minInterTT = ratio;
      maxInterTT = ratio;

    } else {

      if (minInterTT > ratio)
        minInterTT = ratio;
      if (maxInterTT < ratio)
        maxInterTT = ratio;

    }
    sumInterTT += ratio;
    ++countInterTT;

  }

// 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;

  if (paramP.type == FrameCuboid &&
      paramQ.type == FrameCuboid) {

    if (countNoInterCC == 0) {

      minNoInterCC = ratio;
      maxNoInterCC = ratio;

    } else {

      if (minNoInterCC > ratio)
        minNoInterCC = ratio;
      if (maxNoInterCC < ratio)
```

```
        maxNoInterCC = ratio;

    }
    sumNoInterCC += ratio;
    ++countNoInterCC;

  } else if (paramP.type == FrameCuboid &&
             paramQ.type == FrameTetrahedron) {

    if (countNoInterCT == 0) {

      minNoInterCT = ratio;
      maxNoInterCT = ratio;

    } else {

      if (minNoInterCT > ratio)
        minNoInterCT = ratio;
      if (maxNoInterCT < ratio)
        maxNoInterCT = ratio;

    }
    sumNoInterCT += ratio;
    ++countNoInterCT;

  } else if (paramP.type == FrameTetrahedron &&
             paramQ.type == FrameCuboid) {

    if (countNoInterTC == 0) {

      minNoInterTC = ratio;
      maxNoInterTC = ratio;

    } else {

      if (minNoInterTC > ratio)
        minNoInterTC = ratio;
      if (maxNoInterTC < ratio)
        maxNoInterTC = ratio;

    }
    sumNoInterTC += ratio;
    ++countNoInterTC;

  } else if (paramP.type == FrameTetrahedron &&
             paramQ.type == FrameTetrahedron) {

    if (countNoInterTT == 0) {

      minNoInterTT = ratio;
      maxNoInterTT = ratio;

    } else {

      if (minNoInterTT > ratio)
        minNoInterTT = ratio;
      if (maxNoInterTT < ratio)
        maxNoInterTT = ratio;

    }
    sumNoInterTT += ratio;
    ++countNoInterTT;
```

```
        }
      }

    // 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 Qualify2DDynamic(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Loop on runs
  for (int iRun = 0;
       iRun < NB_RUNS;
       ++iRun) {

    // Ratio intersection/no intersection for the displayed results
    double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

    // Initialize counters
    minInter = 0.0;
    maxInter = 0.0;
    sumInter = 0.0;
    countInter = 0;
    minNoInter = 0.0;
    maxNoInter = 0.0;
    sumNoInter = 0.0;
    countNoInter = 0;

    minInterCC = 0.0;
    maxInterCC = 0.0;
    sumInterCC = 0.0;
    countInterCC = 0;
    minNoInterCC = 0.0;
    maxNoInterCC = 0.0;
    sumNoInterCC = 0.0;
    countNoInterCC = 0;

    minInterCT = 0.0;
    maxInterCT = 0.0;
    sumInterCT = 0.0;
    countInterCT = 0;
```

```
minNoInterCT = 0.0;
maxNoInterCT = 0.0;
sumNoInterCT = 0.0;
countNoInterCT = 0;

minInterTC = 0.0;
maxInterTC = 0.0;
sumInterTC = 0.0;
countInterTC = 0;
minNoInterTC = 0.0;
maxNoInterTC = 0.0;
sumNoInterTC = 0.0;
countNoInterTC = 0;

minInterTT = 0.0;
maxInterTT = 0.0;
sumInterTT = 0.0;
countInterTT = 0;
minNoInterTT = 0.0;
maxNoInterTT = 0.0;
sumNoInterTT = 0.0;
countNoInterTT = 0;

// Declare two variables to memozie the arguments to the
// Qualification function
Param2DTime paramP;
Param2DTime paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

  // Create two random Frame definitions
  Param2DTime* 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;
      param->speed[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
      Qualification2DDynamic(
        paramP,
        paramQ);

    }

  }

  // Display the results
  if (iRun == 0) {

    printf("percPairInter\t");
    printf("countInter\tcountNoInter\t");
    printf("minInter\tavgInter\tmaxInter\t");
    printf("minNoInter\tavgNoInter\tmaxNoInter\t");
    printf("minTotal\tavgTotal\tmaxTotal\t");

    printf("countInterCC\tcountNoInterCC\t");
    printf("minInterCC\tavgInterCC\tmaxInterCC\t");
    printf("minNoInterCC\tavgNoInterCC\tmaxNoInterCC\t");
    printf("minTotalCC\tavgTotalCC\tmaxTotalCC\t");

    printf("countInterCT\tcountNoInterCT\t");
    printf("minInterCT\tavgInterCT\tmaxInterCT\t");
    printf("minNoInterCT\tavgNoInterCT\tmaxNoInterCT\t");
    printf("minTotalCT\tavgTotalCT\tmaxTotalCT\t");

    printf("countInterTC\tcountNoInterTC\t");
    printf("minInterTC\tavgInterTC\tmaxInterTC\t");
    printf("minNoInterTC\tavgNoInterTC\tmaxNoInterTC\t");
    printf("minTotalTC\tavgTotalTC\tmaxTotalTC\t");

    printf("countInterTT\tcountNoInterTT\t");
    printf("minInterTT\tavgInterTT\tmaxInterTT\t");
    printf("minNoInterTT\tavgNoInterTT\tmaxNoInterTT\t");
    printf("minTotalTT\tavgTotalTT\tmaxTotalTT\n");

  }

  printf("%.1f\t", ratioInter);

  printf("%lu\t%lu\t", 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 =
```

```
      ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
    printf("%f\t%f\t%f\t",
      (minNoInter < minInter ? minNoInter : minInter),
      avg,
      (maxNoInter > maxInter ? maxNoInter : maxInter));

    printf("%lu\t%lu\t", countInterCC, countNoInterCC);
    double avgInterCC = sumInterCC / (double)countInterCC;
    printf("%f\t%f\t%f\t", minInterCC, avgInterCC, maxInterCC);
    double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
    printf("%f\t%f\t%f\t", minNoInterCC, avgNoInterCC, maxNoInterCC);
    double avgCC =
      ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
    printf("%f\t%f\t%f\t",
      (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
      avgCC,
      (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));

    printf("%lu\t%lu\t", countInterCT, countNoInterCT);
    double avgInterCT = sumInterCT / (double)countInterCT;
    printf("%f\t%f\t%f\t", minInterCT, avgInterCT, maxInterCT);
    double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
    printf("%f\t%f\t%f\t", minNoInterCT, avgNoInterCT, maxNoInterCT);
    double avgCT =
      ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
    printf("%f\t%f\t%f\t",
      (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
      avgCT,
      (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));

    printf("%lu\t%lu\t", countInterTC, countNoInterTC);
    double avgInterTC = sumInterTC / (double)countInterTC;
    printf("%f\t%f\t%f\t", minInterTC, avgInterTC, maxInterTC);
    double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
    printf("%f\t%f\t%f\t", minNoInterTC, avgNoInterTC, maxNoInterTC);
    double avgTC =
      ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
    printf("%f\t%f\t%f\t",
      (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
      avgTC,
      (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));

    printf("%lu\t%lu\t", countInterTT, countNoInterTT);
    double avgInterTT = sumInterTT / (double)countInterTT;
    printf("%f\t%f\t%f\t", minInterTT, avgInterTT, maxInterTT);
    double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
    printf("%f\t%f\t%f\t", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
      ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    printf("%f\t%f\t%f\n",
      (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
      avgTT,
      (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));

  }

}

int main(int argc, char** argv) {

  Qualify2DDynamic();
```

```
  return 0;
}
```

### 8.1.4 3D dynamic

```
// 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 "fmb3dt.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 100000
// 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

// 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[3];
  double comp[3][3];
  double speed[3];
} Param3DTime;

// 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;

double minInterCC;
double maxInterCC;
double sumInterCC;
unsigned long countInterCC;
double minNoInterCC;
double maxNoInterCC;
double sumNoInterCC;
unsigned long countNoInterCC;

double minInterCT;
double maxInterCT;
```

```
double sumInterCT;
unsigned long countInterCT;
double minNoInterCT;
double maxNoInterCT;
double sumNoInterCT;
unsigned long countNoInterCT;

double minInterTC;
double maxInterTC;
double sumInterTC;
unsigned long countInterTC;
double minNoInterTC;
double maxNoInterTC;
double sumNoInterTC;
unsigned long countNoInterTC;

double minInterTT;
double maxInterTT;
double sumInterTT;
unsigned long countInterTT;
double minNoInterTT;
double maxNoInterTT;
double sumNoInterTT;
unsigned long countNoInterTT;

// 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 Qualification3DDynamic(
        const Param3DTime paramP,
        const Param3DTime paramQ) {

  // Create the two Frames
  Frame3DTime P =
    Frame3DTimeCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.speed,
      paramP.comp);

  Frame3DTime Q =
    Frame3DTimeCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.speed,
      paramQ.comp);

  // Helper variables to loop on the pair (that, tho) and (tho, that)
  Frame3DTime* that = &P;
  Frame3DTime* 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] =
    FMBTestIntersection3DTime(
      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] =
    SATTestIntersection3DTime(
      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");
    Frame3DTimePrint(that);
    printf(" against ");
    Frame3DTimePrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB[0] == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT[0] == 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;

    if (paramP.type == FrameCuboid &&
        paramQ.type == FrameCuboid) {
```

```
  if (countInterCC == 0) {

    minInterCC = ratio;
    maxInterCC = ratio;

  } else {

    if (minInterCC > ratio)
      minInterCC = ratio;
    if (maxInterCC < ratio)
      maxInterCC = ratio;

  }
  sumInterCC += ratio;
  ++countInterCC;

} else if (paramP.type == FrameCuboid &&
           paramQ.type == FrameTetrahedron) {

  if (countInterCT == 0) {

    minInterCT = ratio;
    maxInterCT = ratio;

  } else {

    if (minInterCT > ratio)
      minInterCT = ratio;
    if (maxInterCT < ratio)
      maxInterCT = ratio;

  }
  sumInterCT += ratio;
  ++countInterCT;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameCuboid) {

  if (countInterTC == 0) {

    minInterTC = ratio;
    maxInterTC = ratio;

  } else {

    if (minInterTC > ratio)
      minInterTC = ratio;
    if (maxInterTC < ratio)
      maxInterTC = ratio;

  }
  sumInterTC += ratio;
  ++countInterTC;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameTetrahedron) {

  if (countInterTT == 0) {

    minInterTT = ratio;
    maxInterTT = ratio;
```

```
    } else {

      if (minInterTT > ratio)
        minInterTT = ratio;
      if (maxInterTT < ratio)
        maxInterTT = ratio;

    }
    sumInterTT += ratio;
    ++countInterTT;

  }

// 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;

  if (paramP.type == FrameCuboid &&
      paramQ.type == FrameCuboid) {

    if (countNoInterCC == 0) {

      minNoInterCC = ratio;
      maxNoInterCC = ratio;

    } else {

      if (minNoInterCC > ratio)
        minNoInterCC = ratio;
      if (maxNoInterCC < ratio)
        maxNoInterCC = ratio;

    }
    sumNoInterCC += ratio;
    ++countNoInterCC;

  } else if (paramP.type == FrameCuboid &&
             paramQ.type == FrameTetrahedron) {

    if (countNoInterCT == 0) {

      minNoInterCT = ratio;
      maxNoInterCT = ratio;

    } else {
```

```
        if (minNoInterCT > ratio)
          minNoInterCT = ratio;
        if (maxNoInterCT < ratio)
          maxNoInterCT = ratio;

      }
      sumNoInterCT += ratio;
      ++countNoInterCT;

    } else if (paramP.type == FrameTetrahedron &&
               paramQ.type == FrameCuboid) {

      if (countNoInterTC == 0) {

        minNoInterTC = ratio;
        maxNoInterTC = ratio;

      } else {

        if (minNoInterTC > ratio)
          minNoInterTC = ratio;
        if (maxNoInterTC < ratio)
          maxNoInterTC = ratio;

      }
      sumNoInterTC += ratio;
      ++countNoInterTC;

    } else if (paramP.type == FrameTetrahedron &&
               paramQ.type == FrameTetrahedron) {

      if (countNoInterTT == 0) {

        minNoInterTT = ratio;
        maxNoInterTT = ratio;

      } else {

        if (minNoInterTT > ratio)
          minNoInterTT = ratio;
        if (maxNoInterTT < ratio)
          maxNoInterTT = ratio;

      }
      sumNoInterTT += ratio;
      ++countNoInterTT;

    }
  }

// 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 Qualify3DDynamic(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Loop on runs
  for (int iRun = 0;
       iRun < NB_RUNS;
       ++iRun) {

    // Ratio intersection/no intersection for the displayed results
    double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

    // Initialize counters
    minInter = 0.0;
    maxInter = 0.0;
    sumInter = 0.0;
    countInter = 0;
    minNoInter = 0.0;
    maxNoInter = 0.0;
    sumNoInter = 0.0;
    countNoInter = 0;

    minInterCC = 0.0;
    maxInterCC = 0.0;
    sumInterCC = 0.0;
    countInterCC = 0;
    minNoInterCC = 0.0;
    maxNoInterCC = 0.0;
    sumNoInterCC = 0.0;
    countNoInterCC = 0;

    minInterCT = 0.0;
    maxInterCT = 0.0;
    sumInterCT = 0.0;
    countInterCT = 0;
    minNoInterCT = 0.0;
    maxNoInterCT = 0.0;
    sumNoInterCT = 0.0;
    countNoInterCT = 0;

    minInterTC = 0.0;
    maxInterTC = 0.0;
    sumInterTC = 0.0;
    countInterTC = 0;
    minNoInterTC = 0.0;
    maxNoInterTC = 0.0;
    sumNoInterTC = 0.0;
    countNoInterTC = 0;

    minInterTT = 0.0;
    maxInterTT = 0.0;
```

```
sumInterTT = 0.0;
countInterTT = 0;
minNoInterTT = 0.0;
maxNoInterTT = 0.0;
sumNoInterTT = 0.0;
countNoInterTT = 0;

// Declare two variables to memozie the arguments to the
// Qualification function
Param3DTime paramP;
Param3DTime paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

  // Create two random Frame definitions
  Param3DTime* 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;
      param->speed[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
    Qualification3DDynamic(
      paramP,
      paramQ);

  }

}

// Display the results
if (iRun == 0) {

  printf("percPairInter\t");
  printf("countInter\tcountNoInter\t");
  printf("minInter\tavgInter\tmaxInter\t");
  printf("minNoInter\tavgNoInter\tmaxNoInter\t");
  printf("minTotal\tavgTotal\tmaxTotal\t");

  printf("countInterCC\tcountNoInterCC\t");
  printf("minInterCC\tavgInterCC\tmaxInterCC\t");
  printf("minNoInterCC\tavgNoInterCC\tmaxNoInterCC\t");
  printf("minTotalCC\tavgTotalCC\tmaxTotalCC\t");

  printf("countInterCT\tcountNoInterCT\t");
  printf("minInterCT\tavgInterCT\tmaxInterCT\t");
  printf("minNoInterCT\tavgNoInterCT\tmaxNoInterCT\t");
  printf("minTotalCT\tavgTotalCT\tmaxTotalCT\t");

  printf("countInterTC\tcountNoInterTC\t");
  printf("minInterTC\tavgInterTC\tmaxInterTC\t");
  printf("minNoInterTC\tavgNoInterTC\tmaxNoInterTC\t");
  printf("minTotalTC\tavgTotalTC\tmaxTotalTC\t");

  printf("countInterTT\tcountNoInterTT\t");
  printf("minInterTT\tavgInterTT\tmaxInterTT\t");
  printf("minNoInterTT\tavgNoInterTT\tmaxNoInterTT\t");
  printf("minTotalTT\tavgTotalTT\tmaxTotalTT\n");

}

printf("%.1f\t", ratioInter);

printf("%lu\t%lu\t", 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 =
  ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
printf("%f\t%f\t%f\t",
  (minNoInter < minInter ? minNoInter : minInter),
  avg,
  (maxNoInter > maxInter ? maxNoInter : maxInter));

printf("%lu\t%lu\t", countInterCC, countNoInterCC);
double avgInterCC = sumInterCC / (double)countInterCC;
```

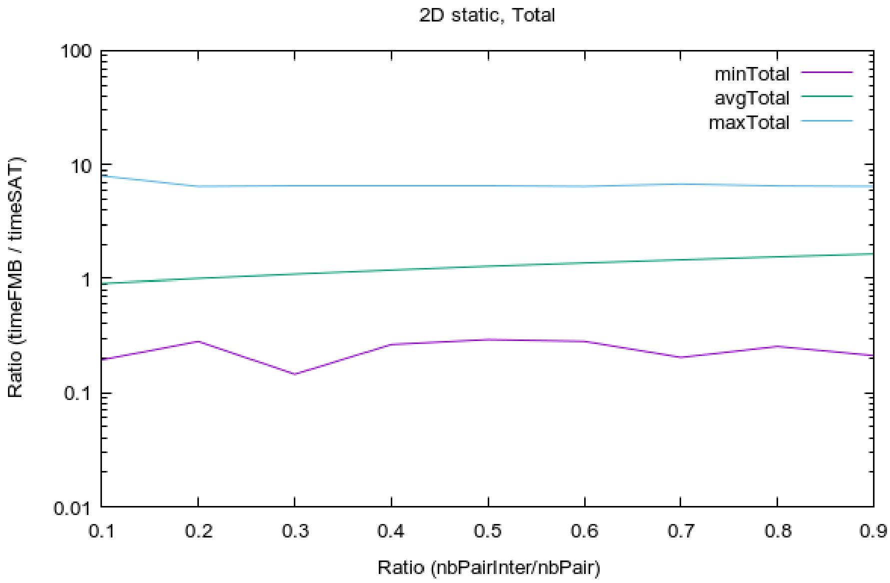
```
    printf("%f\t%f\t%f\t", minInterCC, avgInterCC, maxInterCC);
    double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
    printf("%f\t%f\t%f\t", minNoInterCC, avgNoInterCC, maxNoInterCC);
    double avgCC =
      ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
    printf("%f\t%f\t%f\t",
      (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
      avgCC,
      (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));

    printf("%lu\t%lu\t", countInterCT, countNoInterCT);
    double avgInterCT = sumInterCT / (double)countInterCT;
    printf("%f\t%f\t%f\t", minInterCT, avgInterCT, maxInterCT);
    double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
    printf("%f\t%f\t%f\t", minNoInterCT, avgNoInterCT, maxNoInterCT);
    double avgCT =
      ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
    printf("%f\t%f\t%f\t",
      (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
      avgCT,
      (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));

    printf("%lu\t%lu\t", countInterTC, countNoInterTC);
    double avgInterTC = sumInterTC / (double)countInterTC;
    printf("%f\t%f\t%f\t", minInterTC, avgInterTC, maxInterTC);
    double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
    printf("%f\t%f\t%f\t", minNoInterTC, avgNoInterTC, maxNoInterTC);
    double avgTC =
      ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
    printf("%f\t%f\t%f\t",
      (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
      avgTC,
      (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));

    printf("%lu\t%lu\t", countInterTT, countNoInterTT);
    double avgInterTT = sumInterTT / (double)countInterTT;
    printf("%f\t%f\t%f\t", minInterTT, avgInterTT, maxInterTT);
    double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
    printf("%f\t%f\t%f\t", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
      ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    printf("%f\t%f\t%f\n",
      (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
      avgTT,
      (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));

  }

}

int main(int argc, char** argv) {

  Qualify3DDynamic();

  return 0;
}
```

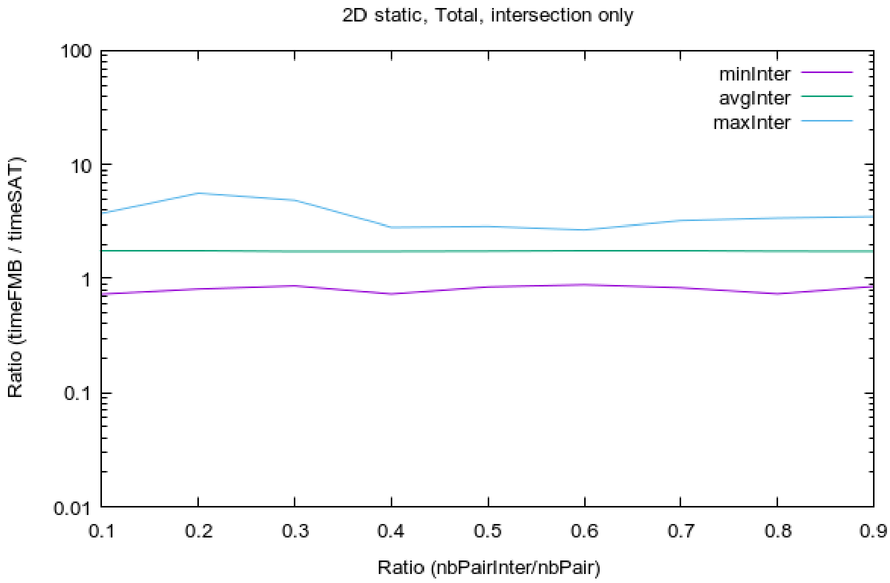
## 8.2 Results

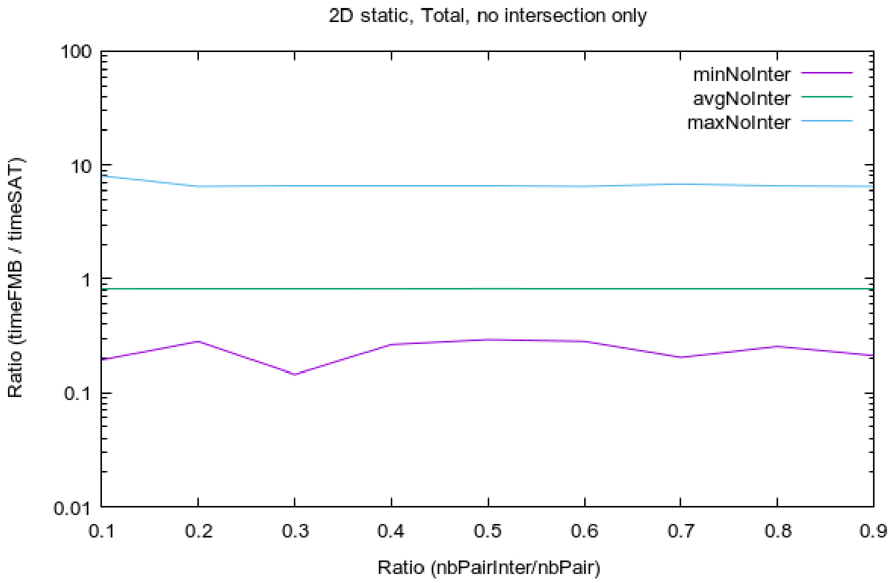
### 8.2.1 2D static

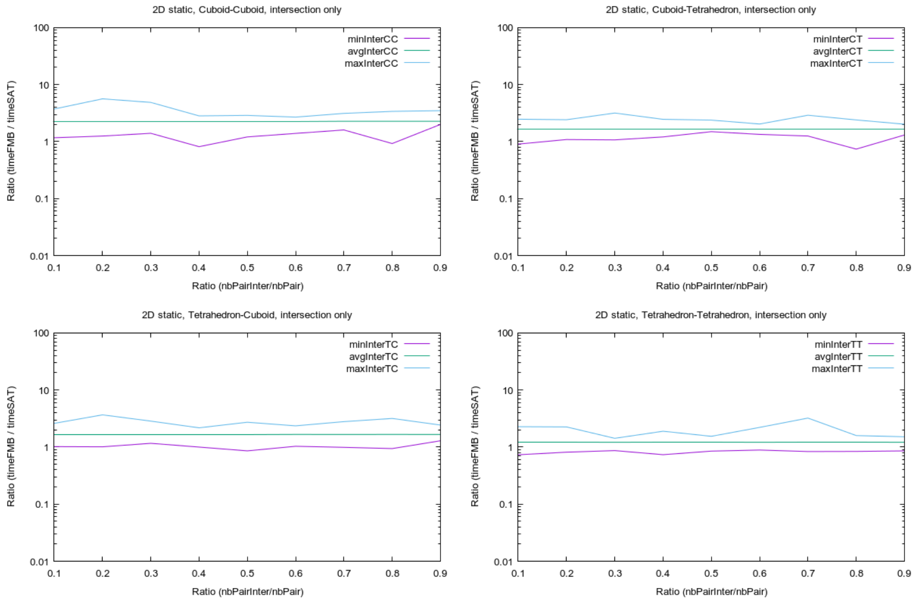
```
percPairInter   countInter      countNoInter    minInter        avgInter
        maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    maxNoInterCC    minTotalCC
    avgTotalCC      maxTotalCC      countInterCT    countNoInterCT
    minInterCT      avgInterCT      maxInterCT      minNoInterCT
    avgNoInterCT    maxNoInterCT    minTotalCT      avgTotalCT
    maxTotalCT      countInterTC    countNoInterTC  minInterTC
    avgInterTC      maxInterTC      minNoInterTC    avgNoInterTC
    maxNoInterTC    minTotalTC      avgTotalTC      maxTotalTC
    countInterTT    countNoInterTT  minInterTT      avgInterTT
    maxInterTT      minNoInterTT    avgNoInterTT    maxNoInterTT
    minTotalTT      avgTotalTT      maxTotalTT
0.1     47084   152908  0.732759        1.736506        3.730769
    0.194030        0.816694        8.000000        0.194030        0.908675
        8.000000        13296   36604   1.166667        2.267757
    3.730769        0.266667        0.773344        6.388889        0.266667
        0.922785        6.388889        11592   38218   0.903846
    1.669039        2.467742        0.285714        0.818122        6.600000
        0.285714        0.903214        6.600000        11784   38452
    1.018692        1.665193        2.600000        0.194030        0.818958
        6.466667        0.194030        0.903581        6.466667
    10412   39634   0.732759        1.213930        2.275362        0.203704
        0.853155        8.000000        0.203704        0.889232
    8.000000
0.2     47292   152692  0.813084        1.736950        5.622642
    0.282051        0.818526        6.466667        0.282051        1.002211
        6.466667        13282   36644   1.250000        2.267540
    5.622642        0.307692        0.775952        3.263158        0.307692
        1.074270        5.622642        11724   38466   1.086022
    1.669223        2.416667        0.324324        0.819762        6.466667
        0.324324        0.989655        6.466667        11976   38166
    1.010870        1.665154        3.666667        0.315789        0.822626
        6.333333        0.315789        0.991132        6.333333
    10310   39416   0.813084        1.213821        2.257143        0.282051
        0.852928        6.214286        0.282051        0.925107
    6.214286
0.3     46710   153290  0.867347        1.732955        4.867925
    0.144737        0.818635        6.533333        0.144737        1.092931
        6.533333        12870   37190   1.397727        2.267700
    4.867925        0.361111        0.777603        3.650000        0.361111
        1.224632        4.867925        11740   38086   1.070707
    1.669222        3.166667        0.144737        0.813867        6.533333
        0.144737        1.070474        6.533333        11870   38492
    1.163043        1.664309        2.852459        0.289474        0.823176
        6.333333        0.289474        1.075516        6.333333
    10230   39522   0.867347        1.213002        1.424242        0.307692
        0.857419        6.214286        0.307692        0.964094
    6.214286
0.4     46890   153100  0.735043        1.735427        2.826923
    0.265306        0.817451        6.533333        0.265306        1.184641
        6.533333        13138   36634   0.813793        2.266767
    2.826923        0.317073        0.778415        3.263158        0.317073
        1.373756        3.263158        11606   38664   1.207792
    1.669974        2.457627        0.265306        0.815315        6.400000
        0.265306        1.157179        6.400000        11814   38424
    1.000000        1.665097        2.176471        0.315789        0.820690
        6.533333        0.315789        1.158453        6.533333
    10332   39378   0.735043        1.213725        1.895522        0.287356
        0.852703        6.285714        0.287356        0.997112
    6.285714
```

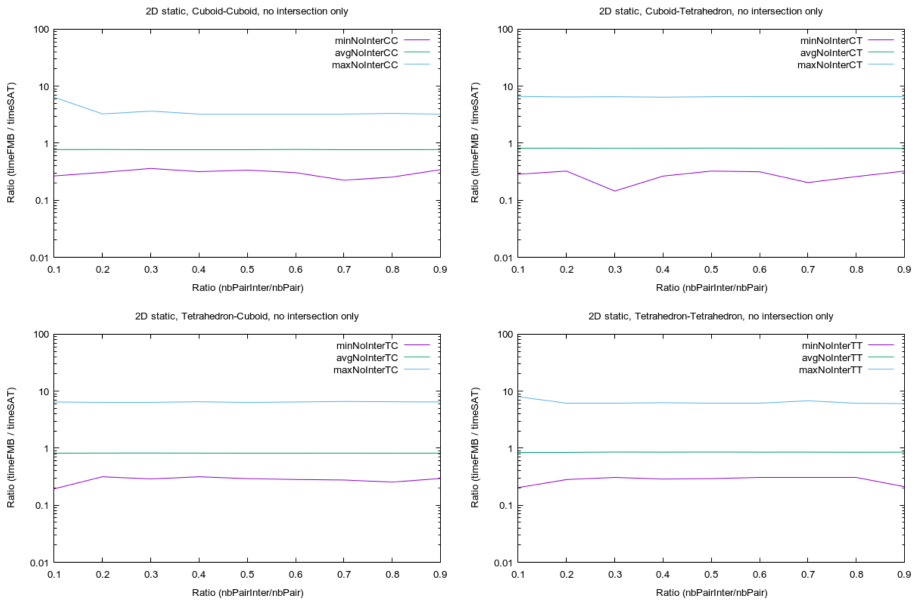
```
0.5     46946   153048  0.848485        1.740979        2.882353
    0.292683        0.819773        6.533333        0.292683        1.280376
        6.533333        13272   37092   1.207547        2.267440
    2.882353        0.337838        0.777748        3.263158        0.337838
        1.522594        3.263158        11898   37838   1.492063
    1.669217        2.380952        0.324324        0.822510        6.533333
        0.324324        1.245863        6.533333        11848   37834
    0.858407        1.664768        2.724138        0.292683        0.819406
        6.333333        0.292683        1.242087        6.333333
    9928    40284   0.848485        1.214146        1.544118        0.292683
        0.856243        6.214286        0.292683        1.035194
    6.214286
0.6     46752   153242  0.887755        1.737497        2.686275
    0.282609        0.817420        6.466667        0.282609        1.369466
        6.466667        13140   36804   1.390805        2.267842
    2.686275        0.304348        0.777269        3.236842        0.304348
        1.671613        3.236842        11572   38378   1.337500
    1.669520        2.034483        0.315789        0.817531        6.466667
        0.315789        1.328724        6.466667        11886   38974
    1.032609        1.665424        2.355932        0.282609        0.819185
        6.466667        0.282609        1.326929        6.466667
    10154   39086   0.887755        1.213030        2.212121        0.307692
        0.853357        6.214286        0.307692        1.069161
    6.214286
0.7     47040   152952  0.834951        1.736977        3.220588
    0.203704        0.816886        6.785714        0.203704        1.460950
        6.785714        13184   36418   1.600000        2.268905
    3.127273        0.224490        0.774422        3.236842        0.224490
        1.820560        3.236842        11798   38402   1.252874
    1.668967        2.900000        0.203704        0.814436        6.466667
        0.203704        1.412608        6.466667        11806   38294
    0.989362        1.664645        2.790323        0.276596        0.821354
        6.600000        0.276596        1.411657        6.600000
    10252   39838   0.834951        1.214486        3.220588        0.307692
        0.853772        6.785714        0.307692        1.106271
    6.785714
0.8     46496   153490  0.738095        1.739313        3.389610
    0.254902        0.815649        6.533333        0.254902        1.554581
        6.533333        13206   37064   0.923077        2.268074
    3.389610        0.254902        0.774775        3.325000        0.254902
        1.969414        3.389610        11544   38556   0.738095
    1.669876        2.400000        0.260000        0.818722        6.466667
        0.260000        1.499645        6.466667        11612   38290
    0.940000        1.665812        3.175439        0.254902        0.817354
        6.533333        0.254902        1.496120        6.533333
    10134   39580   0.838384        1.213585        1.592593        0.307692
        0.849283        6.142857        0.307692        1.140725
    6.142857
0.9     46930   153068  0.857143        1.738010        3.490566
    0.211538        0.816498        6.466667        0.211538        1.645858
        6.466667        13198   36650   2.000000        2.268877
    3.490566        0.342857        0.773279        3.230769        0.342857
        2.119317        3.490566        11812   38618   1.305556
    1.669322        2.017241        0.324324        0.816701        6.466667
        0.324324        1.584060        6.466667        11766   38186
    1.289157        1.664476        2.433333        0.294118        0.819339
        6.466667        0.294118        1.579962        6.466667
    10154   39614   0.857143        1.213108        1.514286        0.211538
        0.853547        6.071429        0.211538        1.177152
    6.071429
```


2D static, Total
minTotal
avgTotal
maxTotal
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)


2D static, Total, intersection only
minInter
avgInter
maxInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)


2D static, Total, no intersection only
minNoInter
avgNoInter
maxNoInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

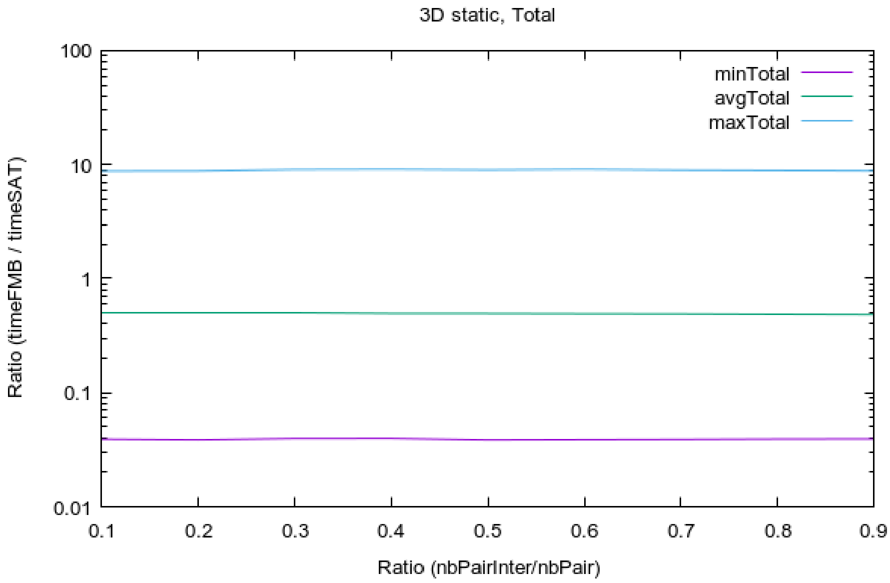


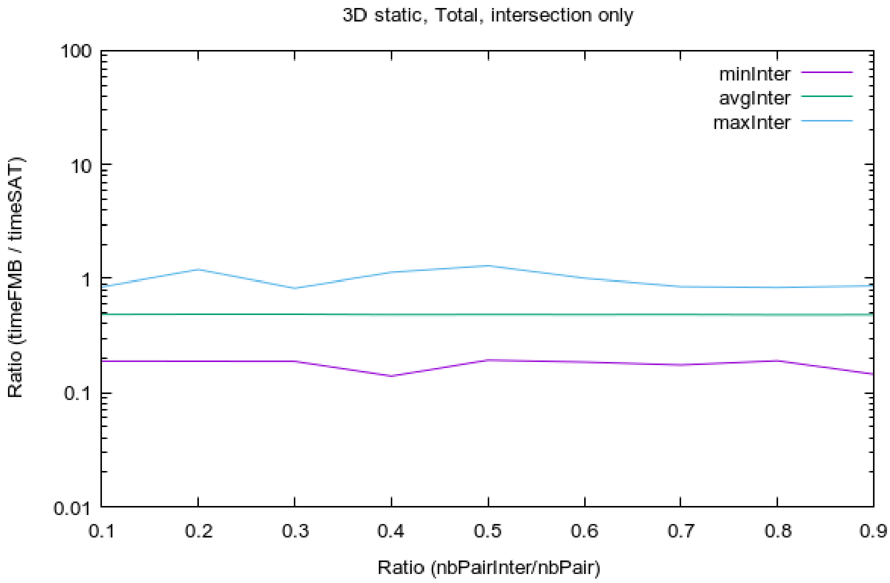



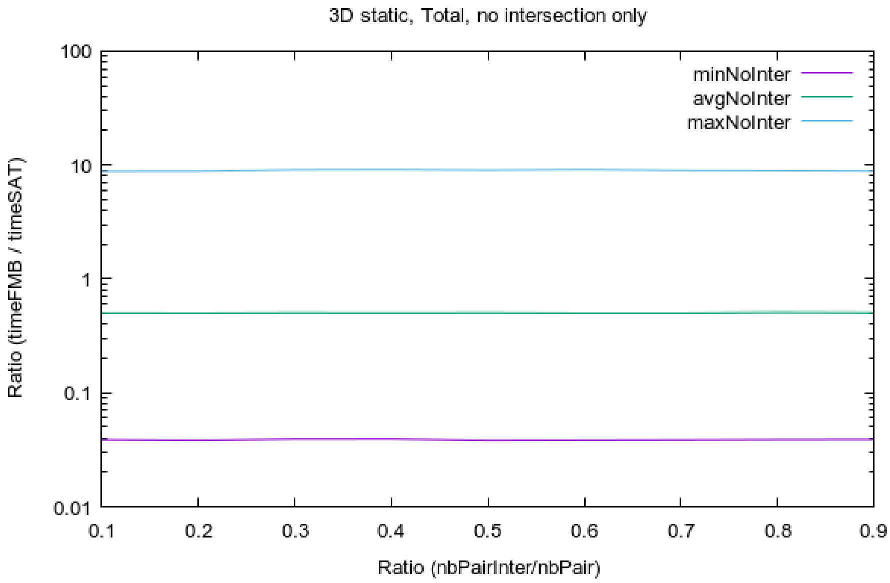

### 8.2.2 3D static

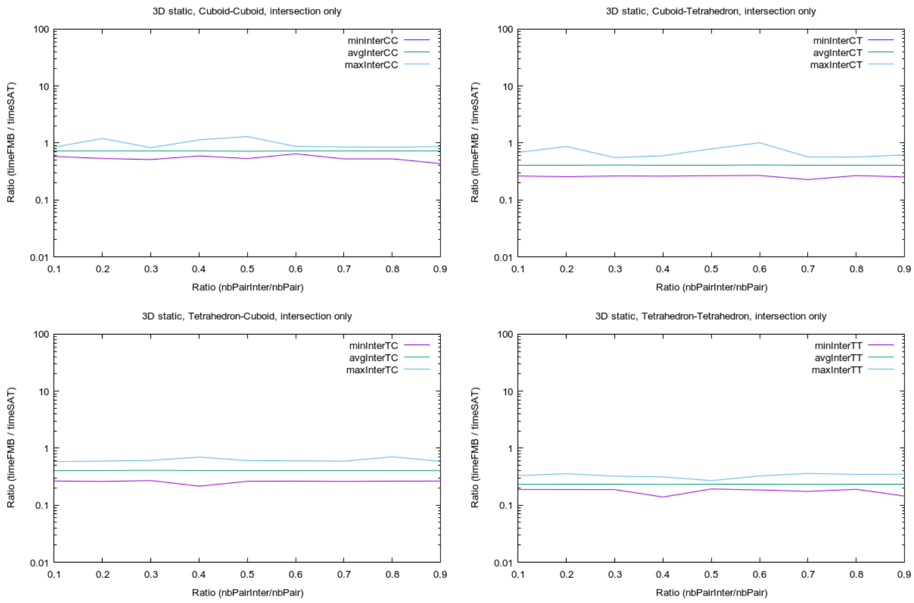
```
percPairInter   countInter      countNoInter    minInter        avgInter
        maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    maxNoInterCC    minTotalCC
    avgTotalCC      maxTotalCC      countInterCT    countNoInterCT
    minInterCT      avgInterCT      maxInterCT      minNoInterCT
    avgNoInterCT    maxNoInterCT    minTotalCT      avgTotalCT
    maxTotalCT      countInterTC    countNoInterTC  minInterTC
    avgInterTC      maxInterTC      minNoInterTC    avgNoInterTC
    maxNoInterTC    minTotalTC      avgTotalTC      maxTotalTC
    countInterTT    countNoInterTT  minInterTT      avgInterTT
    maxInterTT      minNoInterTT    avgNoInterTT    maxNoInterTT
    minTotalTT      avgTotalTT      maxTotalTT
0.1     31568   168432  0.188995        0.484271        0.847670
    0.038849        0.503557        8.781250        0.038849        0.501628
        8.781250        10716   39266   0.579942        0.714376
    0.847670        0.062366        0.359545        3.486301        0.062366
        0.395028        3.486301        7664    42412   0.262613
    0.409951        0.683702        0.045151        0.487978        8.781250
        0.045151        0.480175        8.781250        8042    42196
    0.266120        0.409912        0.582621        0.043956        0.488455
        8.718750        0.043956        0.480601        8.718750
    5146    44558   0.188995        0.231991        0.329949        0.038849
        0.659596        8.208333        0.038849        0.616835
    8.208333
0.2     31716   168284  0.188609        0.483069        1.197441
    0.038298        0.502027        8.812500        0.038298        0.498235
        8.812500        10646   39484   0.534506        0.714731
    1.197441        0.062366        0.355441        2.787671        0.062366
        0.427299        2.787671        7788    42554   0.255875
    0.410237        0.865942        0.044800        0.490341        8.625000
        0.044800        0.474320        8.625000        8082    41948
    0.260870        0.409896        0.593245        0.045089        0.489081
        8.812500        0.045089        0.473244        8.812500
    5200    44298   0.188609        0.231591        0.357326        0.038298
        0.656167        8.291667        0.038298        0.571252
    8.291667
0.3     31388   168612  0.188024        0.482573        0.826549
    0.039359        0.504699        9.064516        0.039359        0.498061
        9.064516        10606   39566   0.510127        0.714160
    0.826549        0.064018        0.353973        2.748299        0.064018
        0.462029        2.748299        7828    42044   0.263226
    0.409572        0.552749        0.044374        0.490852        9.064516
        0.044374        0.466468        9.064516        7672    42308
    0.271233        0.409541        0.610698        0.044657        0.494305
        8.687500        0.044657        0.468876        8.687500
    5282    44694   0.188024        0.231823        0.323106        0.039359
        0.660997        8.208333        0.039359        0.532244
    8.208333
0.4     31282   168718  0.139262        0.481770        1.133545
    0.039416        0.503490        9.096774        0.039416        0.494802
        9.096774        10448   39568   0.591528        0.714639
    1.133545        0.062780        0.355770        2.945946        0.062780
        0.499318        2.945946        7980    41980   0.261628
    0.409862        0.596774        0.045902        0.489760        8.437500
        0.045902        0.457801        8.437500        7614    42704
    0.215993        0.409875        0.698592        0.045234        0.488793
        9.096774        0.045234        0.457226        9.096774
    5240    44466   0.139262        0.231428        0.313602        0.039416
        0.662014        8.291667        0.039416        0.489780
    8.291667
```

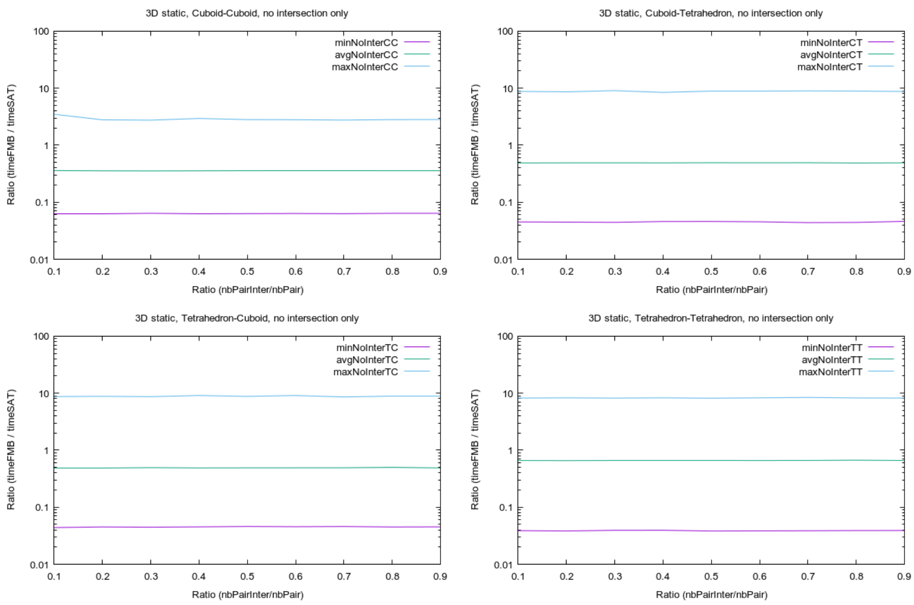
```
0.5     31082   168918  0.192494        0.483474        1.295374
    0.038244        0.504941        9.000000        0.038244        0.494208
            9.000000        10418   39572   0.531081        0.714952
    1.295374        0.063181        0.357895        2.810811        0.063181
            0.536424        2.810811        7838    42338   0.265857
    0.409974        0.789550        0.045977        0.492918        9.000000
            0.045977        0.451446        9.000000        7800    42682
    0.262803        0.410154        0.606461        0.045977        0.490113
            8.781250        0.045977        0.450134        8.781250
    5026    44326   0.192494        0.232071        0.269430        0.038244
            0.661978        8.166667        0.038244        0.447025
    8.166667
0.6     31396   168604  0.185423        0.482966        1.005797
    0.038462        0.505565        9.096774        0.038462        0.492006
            9.096774        10516   39108   0.644845        0.714780
    0.868132        0.063457        0.358111        2.800000        0.063457
            0.572112        2.800000        7968    41942   0.268226
    0.410383        1.005797        0.045381        0.491348        9.000000
            0.045381        0.442769        9.000000        7748    42384
    0.264822        0.410183        0.598820        0.045528        0.491323
            9.096774        0.045528        0.442639        9.096774
    5164    45170   0.185423        0.232095        0.326478        0.038462
            0.659796        8.291667        0.038462        0.403175
    8.291667
0.7     31930   168070  0.174699        0.483689        0.853357
    0.038627        0.505637        8.967742        0.038627        0.490274
            8.967742        10796   39320   0.524899        0.714193
    0.853357        0.063043        0.358114        2.756757        0.063043
            0.607369        2.756757        7964    41988   0.227222
    0.410245        0.569069        0.043887        0.492371        8.967742
            0.043887        0.434883        8.967742        7936    42042
    0.260700        0.409905        0.591508        0.045902        0.491772
            8.593750        0.045902        0.434465        8.593750
    5234    44720   0.174699        0.231864        0.361702        0.038627
            0.660837        8.434783        0.038627        0.360556
    8.434783
0.8     31426   168574  0.190132        0.480869        0.838828
    0.038961        0.507928        8.906250        0.038961        0.486281
            8.906250        10474   39564   0.526104        0.714300
    0.838828        0.063927        0.357892        2.813793        0.063927
            0.643018        2.813793        7792    42090   0.267030
    0.409815        0.565217        0.044234        0.486977        8.906250
            0.044234        0.425248        8.906250        7766    42160
    0.264666        0.410111        0.706745        0.045016        0.499477
            8.838710        0.045016        0.427984        8.838710
    5394    44760   0.190132        0.232111        0.347630        0.038961
            0.668207        8.259259        0.038961        0.319330
    8.259259
0.9     31336   168664  0.144703        0.481202        0.867647
    0.039074        0.504622        8.843750        0.039074        0.483544
            8.843750        10416   39008   0.435466        0.714037
    0.867647        0.064018        0.357324        2.816327        0.064018
            0.678366        2.816327        7824    42532   0.253503
    0.409743        0.616864        0.046129        0.490516        8.750000
            0.046129        0.417820        8.750000        7864    42028
    0.265499        0.409694        0.588323        0.045226        0.488700
            8.843750        0.045226        0.417594        8.843750
    5232    45096   0.144703        0.232012        0.348925        0.039074
            0.660177        8.208333        0.039074        0.274828
    8.208333
```


3D static, Total
minTotal
avgTotal
maxTotal
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


3D static, Total, intersection only
minInter
avgInter
maxInter
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


3D static, Total, no intersection only
minNoInter
avgNoInter
maxNoInter
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

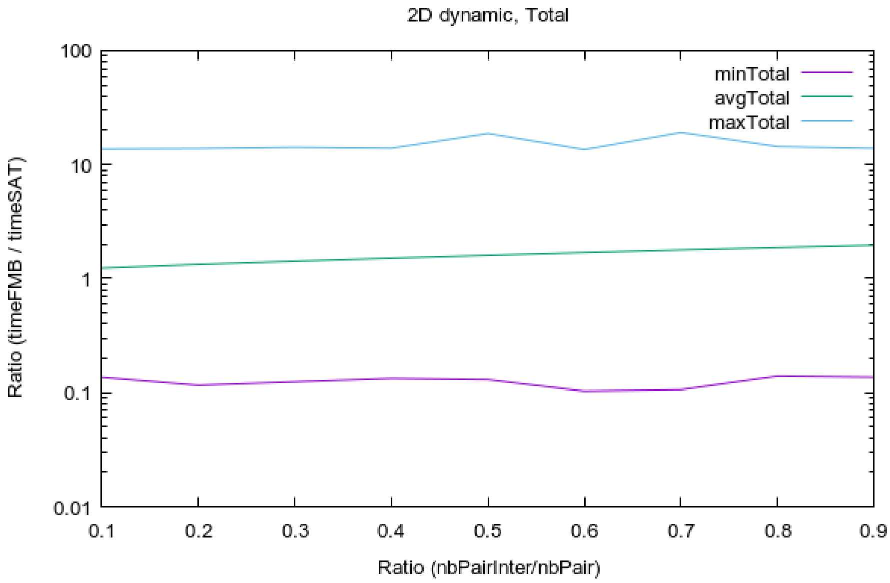


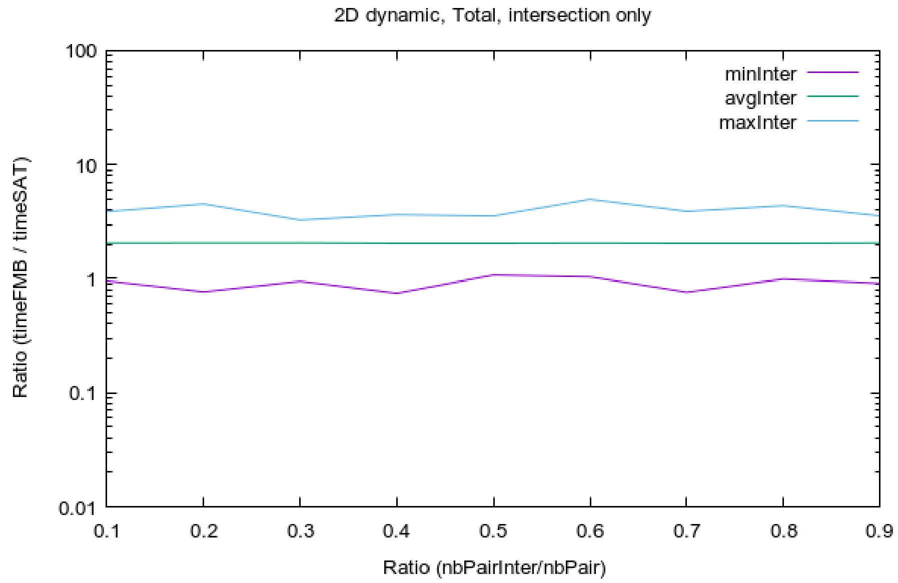



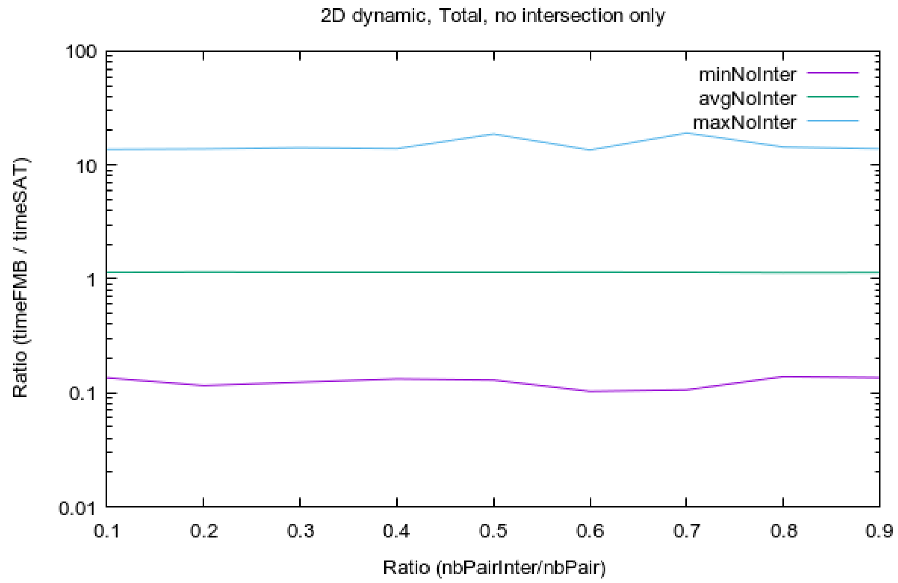

### 8.2.3 2D dynamic

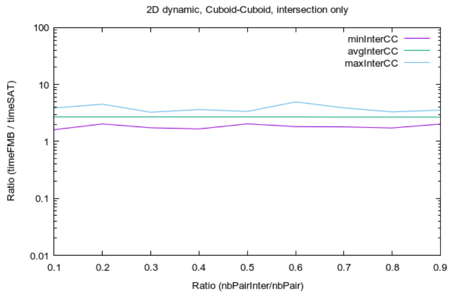
```
percPairInter   countInter      countNoInter    minInter        avgInter
        maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    maxNoInterCC    minTotalCC
    avgTotalCC      maxTotalCC      countInterCT    countNoInterCT
    minInterCT      avgInterCT      maxInterCT      minNoInterCT
    avgNoInterCT    maxNoInterCT    minTotalCT      avgTotalCT
    maxTotalCT      countInterTC    countNoInterTC  minInterTC
    avgInterTC      maxInterTC      minNoInterTC    avgNoInterTC
    maxNoInterTC    minTotalTC      avgTotalTC      maxTotalTC
    countInterTT    countNoInterTT  minInterTT      avgInterTT
    maxInterTT      minNoInterTT    avgNoInterTT    maxNoInterTT
    minTotalTT      avgTotalTT      maxTotalTT
0.1     74174   125820  0.953307        2.061856        3.869231
    0.135922        1.142155        13.760000       0.135922        1.234125
        13.760000       19882   30074   1.608295        2.675634
    3.869231        0.173554        1.154897        13.538462       0.173554
        1.306971        13.538462       18612   31316   1.117886
    1.998857        3.099291        0.148148        1.116133
    13.760000       0.148148        1.204405        13.760000       18362
    31230   1.142857        1.995607        2.900709        0.146789
    1.152028        13.666667       0.146789        1.236386
    13.666667       17318   33200   0.953307        1.495156        2.461078
        0.135922        1.145869        10.458333       0.135922
    1.180798        10.458333
0.2     74328   125660  0.766355        2.065718        4.511450
    0.115942        1.150277        13.880000       0.115942        1.333365
        13.880000       20246   30176   2.029070        2.674944
    4.511450        0.177083        1.133148        13.714286       0.177083
        1.441507        13.714286       18410   31628   1.340000
    1.999344        3.094203        0.146789        1.163951
    13.625000       0.146789        1.331029        13.625000       18460
    31272   1.248848        1.995984        2.631206        0.115942
    1.158049        13.880000       0.115942        1.325636
    13.880000       17212   32584   0.766355        1.494886        2.505263
        0.147826        1.145407        10.416667       0.147826
    1.215302        10.416667
0.3     74508   125482  0.946154        2.066441        3.263566
    0.124088        1.146541        14.217391       0.124088        1.422511
        14.217391       20108   29998   1.740000        2.675652
    3.263566        0.166667        1.118448        13.538462       0.166667
        1.585609        13.538462       18628   31376   1.376289
    1.999141        2.926471        0.146789        1.160402
    14.217391       0.146789        1.412024        14.217391       18884
    31172   1.421053        1.995321        2.771429        0.124088
    1.157288        13.541667       0.124088        1.408698
    13.541667       16888   32936   0.946154        1.494831        1.860606
        0.156250        1.148753        10.583333       0.156250
    1.252576        10.583333
0.4     74592   125400  0.743827        2.062232        3.635659
    0.132743        1.143175        13.958333       0.132743        1.510798
        13.958333       19964   30348   1.660232        2.675354
    3.635659        0.186813        1.119739        13.920000       0.186813
        1.741985        13.920000       18600   31136   1.340741
    1.999622        2.693431        0.138614        1.132257
    13.666667       0.138614        1.479203        13.666667       18692
    31200   1.283721        1.995636        3.201878        0.132743
    1.155133        13.958333       0.132743        1.491334
    13.958333       17336   32716   0.743827        1.495145        2.909639
        0.163265        1.163904        10.807692       0.163265
    1.296400        10.807692
```

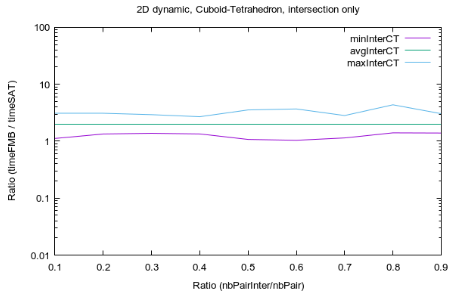
```
0.5     74658   125330  1.073930        2.060883        3.541096
    0.129771        1.142624        18.785714       0.129771        1.601753
            18.785714       19916   29950   2.035088        2.676320
    3.361538        0.180723        1.124233        18.785714       0.180723
            1.900276        18.785714       18416   31678   1.073930
    1.999509        3.541096        0.146789        1.151864
    14.086957       0.146789        1.575687        14.086957       18844
    31642   1.304762        1.995443        2.730496        0.146789
    1.139153        14.555556       0.146789        1.567298
    14.555556       17482   32060   1.199005        1.494952        1.921212
            0.129771        1.154099        10.695652       0.129771
    1.324526        10.695652
0.6     74540   125454  1.038136        2.059363        4.962406
    0.103030        1.147068        13.583333       0.103030        1.694445
            13.583333       19762   30050   1.825000        2.676677
    4.962406        0.139344        1.158353        13.576923       0.139344
            2.069347        13.576923       18488   31506   1.038136
    2.000517        3.687831        0.148148        1.132695
    13.583333       0.148148        1.653388        13.583333       18700
    31228   1.372449        1.995923        3.534247        0.127119
    1.128504        13.583333       0.127119        1.648956
    13.583333       17590   32670   1.065502        1.495115        1.927273
            0.103030        1.168294        13.148148       0.103030
    1.364386        13.148148
0.7     74624   125364  0.760606        2.063555        3.890173
    0.106250        1.145570        19.214286       0.106250        1.788160
            19.214286       20164   30288   1.802521        2.675364
    3.890173        0.190476        1.125170        19.214286       0.190476
            2.210306        19.214286       18586   31510   1.142857
    1.999560        2.824242        0.106250        1.141702
    17.851852       0.106250        1.742203        17.851852       18482
    30988   1.268868        1.995224        2.904762        0.146789
    1.160905        14.173913       0.146789        1.744928
    14.173913       17392   32578   0.760606        1.495237        3.243902
            0.145631        1.153691        10.846154       0.145631
    1.392773        10.846154
0.8     74030   125960  0.992000        2.064627        4.364238
    0.139344        1.135396        14.413793       0.139344        1.878781
            14.413793       20064   30328   1.725490        2.675967
    3.307692        0.195122        1.119281        14.413793       0.195122
            2.364630        14.413793       18424   30972   1.403141
    1.999862        4.364238        0.146789        1.129743
    14.166667       0.146789        1.825838        14.166667       18280
    31472   1.277533        1.996164        2.966667        0.146789
    1.137087        13.583333       0.146789        1.824349
    13.583333       17262   33188   0.992000        1.495679        3.852761
            0.139344        1.153794        10.739130       0.139344
    1.427302        10.739130
0.9     74762   125230  0.908257        2.061844        3.550802
    0.136364        1.138984        13.923077       0.136364        1.969558
            13.923077       20034   29820   2.020305        2.674946
    3.550802        0.137931        1.102084        13.538462       0.137931
            2.517660        13.538462       18862   31318   1.391753
    1.999101        3.043165        0.146789        1.132414
    13.791667       0.146789        1.912432        13.791667       18498
    31394   1.131206        1.994555        2.829114        0.136364
    1.160532        13.923077       0.136364        1.911152
    13.923077       17368   32698   0.908257        1.494437        2.798780
            0.139344        1.158239        10.720000       0.139344
    1.460817        10.720000
```

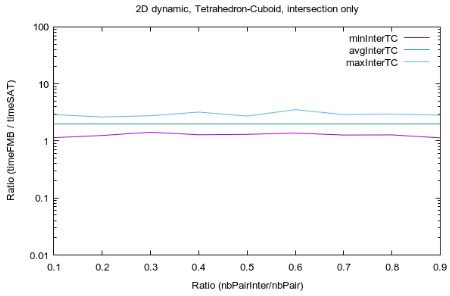

2D dynamic, Total
minTotal
avgTotal
maxTotal
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

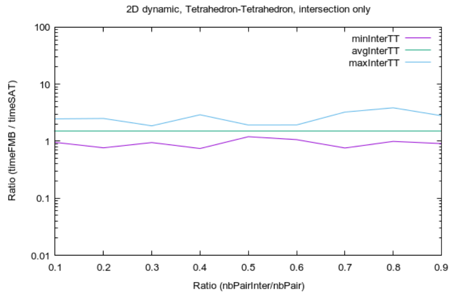

2D dynamic, Total, intersection only
minInter
avgInter
maxInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

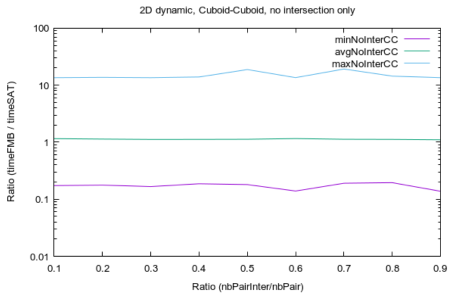

2D dynamic, Total, no intersection only
minNoInter
avgNoInter
maxNoInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

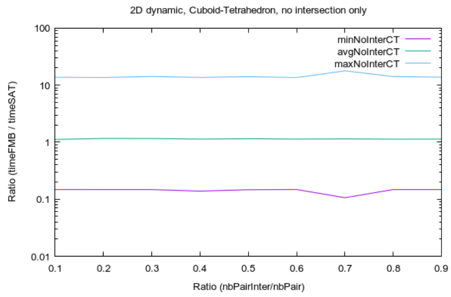


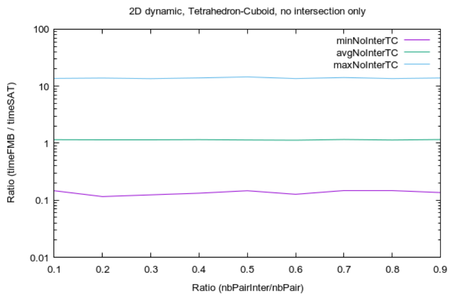



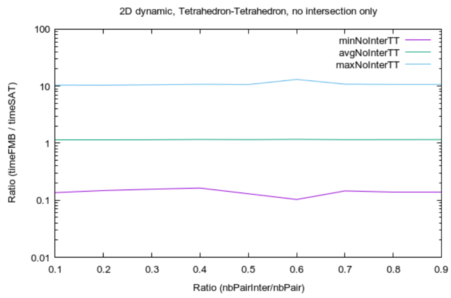



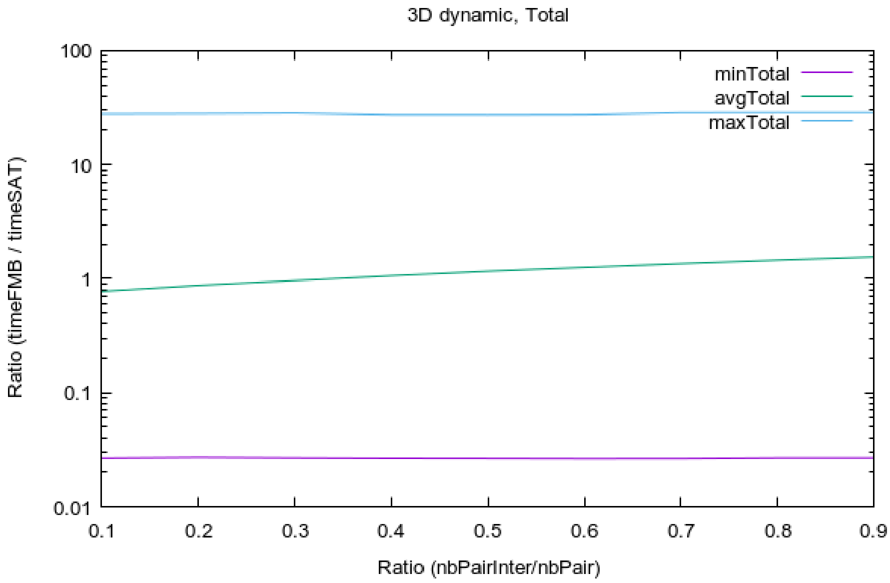



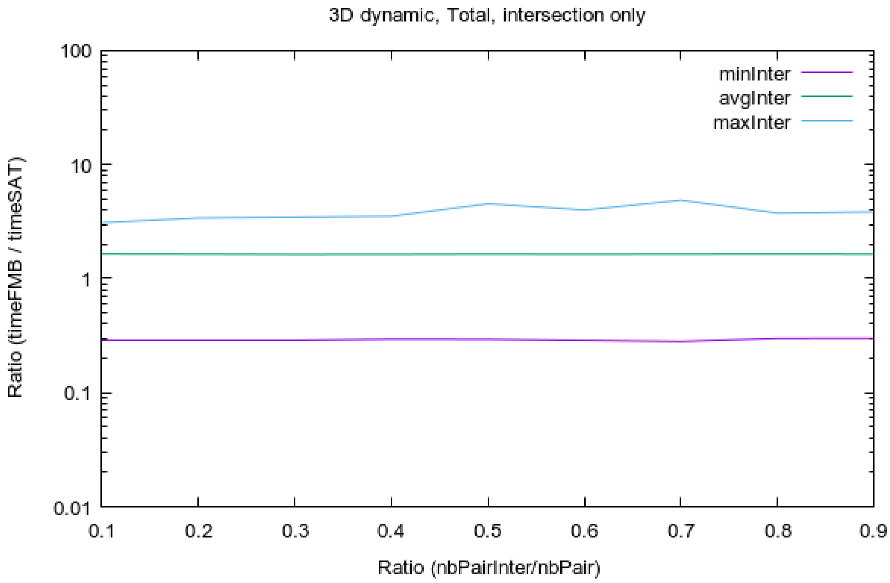

### 8.2.4 3D dynamic

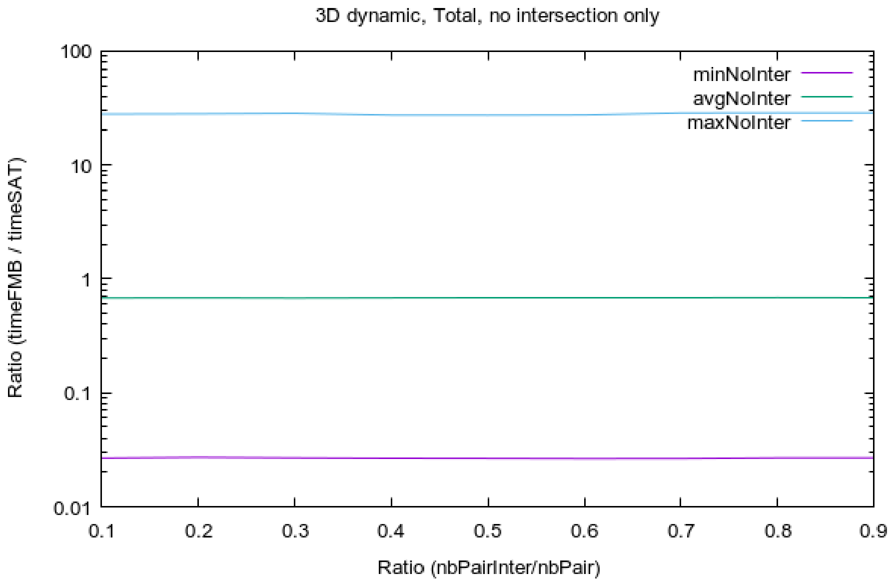
```
percPairInter   countInter      countNoInter    minInter        avgInter
        maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    maxNoInterCC    minTotalCC
    avgTotalCC      maxTotalCC      countInterCT    countNoInterCT
    minInterCT      avgInterCT      maxInterCT      minNoInterCT
    avgNoInterCT    maxNoInterCT    minTotalCT      avgTotalCT
    maxTotalCT      countInterTC    countNoInterTC  minInterTC
    avgInterTC      maxInterTC      minNoInterTC    avgNoInterTC
    maxNoInterTC    minTotalTC      avgTotalTC      maxTotalTC
    countInterTT    countNoInterTT  minInterTT      avgInterTT
    maxInterTT      minNoInterTT    avgNoInterTT    maxNoInterTT
    minTotalTT      avgTotalTT      maxTotalTT
0.1     52992   147008  0.289222        1.645262        3.088929
    0.026688        0.676814        27.972973       0.026688        0.773659
        27.972973       16374   33630   2.055866        2.658200
    3.088929        0.037815        0.603590        13.252212       0.037815
        0.809051        13.252212       13478   37020   0.473251
    1.426485        2.680397        0.027486        0.670883
    22.387755       0.027486        0.746443        22.387755       13088
    36642   0.470426        1.428448        2.583134        0.026688
    0.671253        20.764706       0.026688        0.746973
    20.764706       10052   39716   0.289222        0.570898        0.822473
        0.027157        0.749478        27.972973       0.027157
    0.731620        27.972973
0.2     51846   148154  0.290522        1.640652        3.395413
    0.027092        0.678873        28.108108       0.027092        0.871229
        28.108108       15978   34234   1.828447        2.658938
    3.395413        0.039387        0.595190        15.047826       0.039387
        1.007939        15.047826       12708   37314   0.464358
    1.427126        2.801641        0.027309        0.675123
    22.387755       0.027309        0.825523        22.387755       13054
    36958   0.470907        1.429999        2.580392        0.027344
    0.678087        21.057692       0.027344        0.828470
    21.057692       10106   39648   0.290522        0.571305        0.914305
        0.027092        0.755391        28.108108       0.027092
    0.718574        28.108108
0.3     52046   147954  0.290569        1.630945        3.451671
    0.026814        0.676408        28.388889       0.026814        0.962769
        28.388889       15742   34138   1.941176        2.658088
    3.451671        0.039735        0.595914        13.226667       0.039735
        1.214566        13.226667       13100   36764   0.473901
    1.427939        3.036719        0.027486        0.680578
    22.285714       0.027486        0.904787        22.285714       12954
    36962   0.431818        1.427615        2.821618        0.026814
    0.668215        21.745098       0.026814        0.896035
    21.745098       10250   40090   0.290569        0.569872        0.769397
        0.027309        0.748681        28.388889       0.027309
    0.695039        28.388889
0.4     52380   147620  0.295597        1.634126        3.515625
    0.026625        0.679407        27.378378       0.026625        1.061295
        27.378378       16032   33640   2.129455        2.657113
    3.515625        0.040043        0.587092        13.198238       0.040043
        1.415100        13.198238       13176   36992   0.473140
    1.428340        2.757031        0.027620        0.692006
    22.120000       0.027620        0.986540        22.120000       12830
    36962   0.407386        1.425436        2.739907        0.027553
    0.669286        22.200000       0.027553        0.971746
    22.200000       10342   40026   0.295597        0.569381        0.780453
        0.026625        0.754698        27.378378       0.026625
    0.680571        27.378378
```

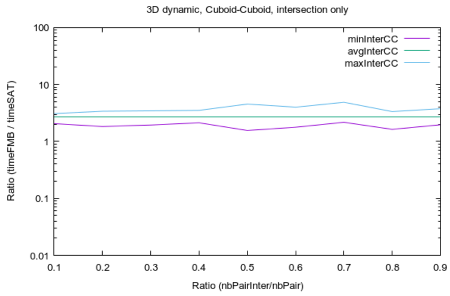
```
0.5     52446   147554  0.294810        1.642163        4.536860
    0.026521        0.676456        27.351351       0.026521        1.159309
            27.351351       15974   33912   1.557248        2.657868
    4.536860        0.039615        0.609089        13.324444       0.039615
            1.633478        13.324444       13642   36824   0.359772
    1.429747        3.036129        0.026984        0.667185
    23.142857       0.026984        1.048466        23.142857       12998
    36988   0.473829        1.427631        2.949298        0.027710
    0.674889        20.700000       0.027710        1.051260
    20.700000       9832    39830   0.294810        0.570295        0.782668
            0.026521        0.743840        27.351351       0.026521
    0.657068        27.351351
0.6     52580   147420  0.288331        1.635602        3.997250
    0.026418        0.676055        27.459459       0.026418        1.251783
            27.459459       16016   33620   1.778186        2.657289
    3.997250        0.040089        0.599354        13.352174       0.040089
            1.834115        13.352174       13086   37174   0.357567
    1.429193        2.563981        0.027178        0.672724
    22.854167       0.027178        1.126605        22.854167       13238
    37038   0.474930        1.427524        3.043171        0.028007
    0.664174        22.183673       0.028007        1.122184
    22.183673       10240   39588   0.288331        0.570393        0.802878
            0.026418        0.755438        27.459459       0.026418
    0.644411        27.459459
0.7     52346   147654  0.282432        1.640549        4.879111
    0.026442        0.676479        28.611111       0.026442        1.351328
            28.611111       16040   34414   2.172337        2.657809
    4.879111        0.039823        0.598944        13.220264       0.039823
            2.040149        13.220264       13092   36852   0.474554
    1.429232        3.339258        0.027157        0.687753
    22.469388       0.027157        1.206788        22.469388       13138
    36578   0.475187        1.430014        4.046765        0.026583
    0.667598        21.134615       0.026583        1.201289
    21.134615       10076   39810   0.282432        0.570255        0.764875
            0.026442        0.741229        28.611111       0.026442
    0.621547        28.611111
0.8     52142   147858  0.300141        1.642869        3.748462
    0.026835        0.683747        28.694444       0.026835        1.451045
            28.694444       15998   34236   1.631609        2.657593
    3.349365        0.039823        0.595301        13.251101       0.039823
            2.245135        13.251101       13006   36684   0.473430
    1.428222        3.748462        0.027778        0.681028
    22.060000       0.027778        1.278783        22.060000       13284
    37080   0.420061        1.427358        2.676242        0.026835
    0.690518        21.877551       0.026835        1.279990
    21.877551       9854    39858   0.300141        0.569295        0.869427
            0.027049        0.755922        28.694444       0.027049
    0.606620        28.694444
0.9     52476   147524  0.300819        1.641245        3.842628
    0.026856        0.680868        28.657143       0.026856        1.545207
            28.657143       16188   34142   1.960784        2.658019
    3.761905        0.039779        0.596012        13.216814       0.039779
            2.451818        13.216814       13144   36900   0.473171
    1.428146        2.635665        0.027353        0.679893
    21.269231       0.027353        1.353321        21.269231       12992
    36714   0.395740        1.426743        3.842628        0.027665
    0.682460        22.448980       0.027665        1.352314
    22.448980       10152   39768   0.300819        0.570348        0.802935
            0.026856        0.753154        28.657143       0.026856
    0.588629        28.657143
```

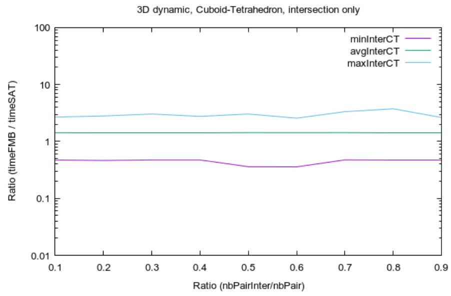


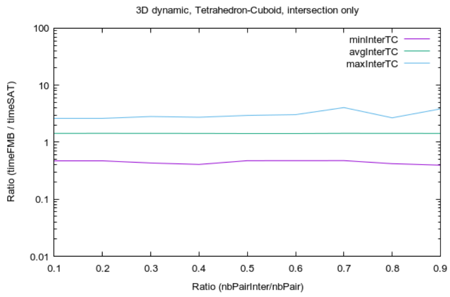
3D dynamic, Total
minTotal
avgTotal
maxTotal
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

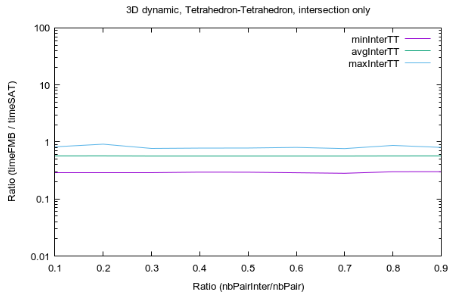

3D dynamic, Total, intersection only
100
10
1
0.1
0.01
Ratio (timeFMB / timeSAT)
minInter
avgInter
maxInter
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

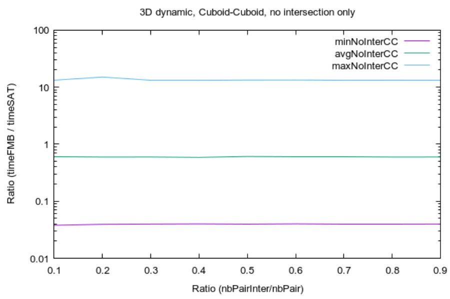

3D dynamic, Total, no intersection only
100
10
1
0.1
0.01
Ratio (timeFMB / timeSAT)
minNoInter
avgNoInter
maxNoInter
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

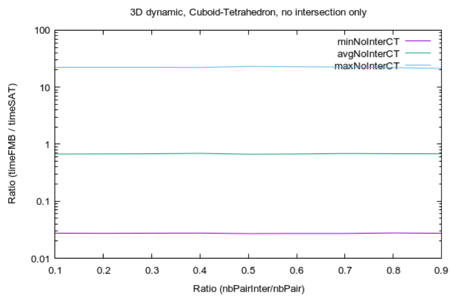

3D dynamic, Cuboid-Cuboid, intersection only
minInterCC
avgInterCC
maxInterCC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9

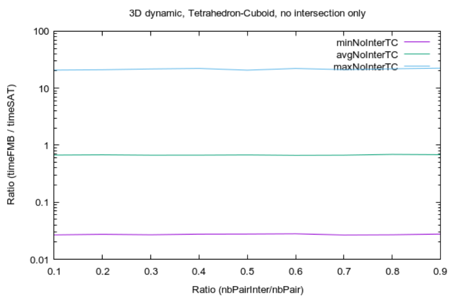

3D dynamic, Cuboid-Tetrahedron, intersection only
minInterCT
avgInterCT
maxInterCT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9

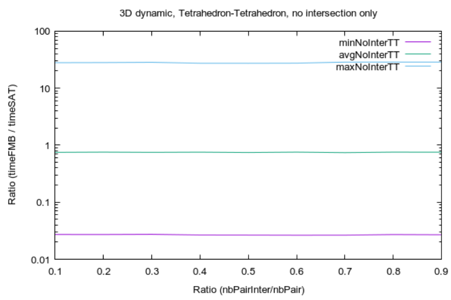

3D dynamic, Tetrahedron-Cuboid, intersection only
minInterTC
avgInterTC
maxInterTC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9


3D dynamic, Tetrahedron-Tetrahedron, intersection only
minInterTT
avgInterTT
maxInterTT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9


3D dynamic, Cuboid-Cuboid, no intersection only
minNoInterCC
avgNoInterCC
maxNoInterCC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9


3D dynamic, Cuboid-Tetrahedron, no intersection only
minNoInterCT
avgNoInterCT
maxNoInterCT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9


3D dynamic, Tetrahedron-Cuboid, no intersection only
minNoInterTC
avgNoInterTC
maxNoInterTC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9


3D dynamic, Tetrahedron-Tetrahedron, no intersection only
minNoInterTT
avgNoInterTT
maxNoInterTT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9

# 9 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 shows that the FMB is 1.2 to 1.8 times slower than the SAT algorithm in the 2D dynamic case. However it is around 2 times faster in the 3D static case, and up to 1.25 times faster in 3D dynamic and up to 1.1 times faster in the 2D static case if the percentage of tested pairs in intersection is less than, respectively, around 40% and 25%.

On one given pair of Frame, the relative speed of the FMB algorithm varies widely, from around 20 times slower to 50 times faster. This is explained by the way the 2 algorithms works: they both make the asumption that the Frames are intersecting and run through a series of tests to try to prove it wrong. This leads to best cases and worst cases for both algorithm: a non interesecting detected right from the first test, or one detected by the last test. These best and worst cases are different for the two algorithm as the tests they performed are completely different. But globally, the FMB algorithm has the advantage.

# 10 Annex

## 10.1 Runtime environment

Results introduce in this paper have been produced by compiling and running the corresponding algorithms in the following environment:

```
    > uname -v 40 18.04.1-Ubuntu SMP Thu Nov 14 12:06:39 UTC 2019
    > lshw -short H/W path Device Class Description =====================================================
system VC65-C1 /0 bus VC65-C1 /0/0 memory 64KiB BIOS /0/2f memory 16GiB System Memory /0/2f/0
memory [empty] /0/2f/1 memory 16GiB SODIMM DDR4 Synchronous 2400 MHz (0.4 ns) /0/39 memory 384KiB
L1 cache /0/3a memory 1536KiB L2 cache /0/3b memory 12MiB L3 cache /0/3c processor Intel(R) Core(TM)
i7-8700T CPU @ 2.40GHz /0/100 bridge 8th Gen Core Processor Host Bridge/DRAM Registers /0/100/2
display Intel Corporation /0/100/12 generic Cannon Lake PCH Thermal Controller /0/100/14 bus
Cannon Lake PCH USB 3.1 xHCI Host Controller /0/100/14/0 usb1 bus xHCI Host Controller /0/100/14/0/5
input ELECOM Wired Keyboard /0/100/14/0/6 input PTZ-630 /0/100/14/0/7 generic USB2.0-CRW /0/100/14/0/e
communication Bluetooth wireless interface /0/100/14/1 usb2 bus xHCI Host Controller /0/100/14.2
memory RAM memory /0/100/14.3 wlo1 network Wireless-AC 9560 [Jefferson Peak] /0/100/16 communication
Cannon Lake PCH HECI Controller /0/100/17 storage Cannon Lake PCH SATA AHCI Controller /0/100/1f
bridge Intel Corporation /0/100/1f.3 multimedia Cannon Lake PCH cAVS /0/100/1f.4 bus Cannon Lake
PCH SMBus Controller /0/100/1f.5 bus Cannon Lake PCH SPI Controller /0/100/1f.6 eno2 network
Ethernet Connection (7) I219-V /0/1 scsi0 storage /0/1/0.0.0 /dev/sda disk 128GB HFS128G39TND-N21
/0/1/0.0.0/1 volume 99MiB Windows FAT volume /0/1/0.0.0/2 /dev/sda2 volume 15MiB reserved partition
/0/1/0.0.0/3 /dev/sda3 volume 83GiB Windows NTFS volume /0/1/0.0.0/4 /dev/sda4 volume 499MiB
Windows NTFS volume /0/1/0.0.0/5 /dev/sda5 volume 35GiB EXT4 volume /0/2 scsi2 storage /0/2/0.0.0
```

/dev/sdb disk 500GB ST500LM034-2GH17 /0/2/0.0.0/1 /dev/sdb1 volume 463GiB EXT4 volume /0/2/0.0.0/2 /dev/sdb2 volume 499MiB Windows FAT volume /0/3 scsi5 storage /0/3/0.0.0 /dev/cdrom disk BD-RE BU50N /1 power To Be Filled By O.E.M.

> lscpu Architecture: x86$_6$4CPUop−mode(s) : 32−bit, 64−bitByteOrder : LittleEndianCPU(s) : 12On − lineCPU(s)list : 0 − 11Thread(s)percore : 2Core(s)persocket : 6Socket(s) : 1NUMAnode(s) : 1VendorID : GenuineIntelCPUfamily : 6Model : 158Modelname : Intel(R)Core(TM)i7−8700TCPU@2.40GHzStepping : 10CPUMHz : 1380.998CPUmaxMHz : 4000.0000CPUminMHz : 800.0000BogoMIPS : 4800.00Virtualization : VT − xL1dcache : 32KL1icache : 32KL2cache : 256KL3cache : 12288KNUMAnode0CPU(s) : 0 − 11Flags : fpuvmedepsetscmsrpaemcecx8apicsepmtrrpgemcacmovpatpse36clflushdtsacpimmxfxsrssesse2sshttmpbesyscallnxpdpe1

> gcc -v Using built-in specs. COLLECT$_G$CC = gccCOLLECT$_L$TO$_W$RAPPER = /usr/lib/gcc/x86$_6$4−linux−gnu/7/lto−wrapperOFFLOAD$_T$ARGET$_N$AMES = nvptx−noneOFFLOAD$_T$ARGET$_D$EFAULT = 1Target : x86$_6$4−linux−gnuConfiguredwith : ../src/configure−v−−with−pkgversion =′ Ubuntu7.4.0−1ubuntu1 18.04.1′ − −with − bugurl = file : ///usr/share/doc/gcc − 7/README.Bugs − −enable − languages = c, ada, c++, go, brig, d, fortran, objc, obj − c++ − −prefix = /usr − −with − gcc − major − version − only − −program − suffix = −7 − −program − prefix = x86$_6$4 − linux − gnu − − − enable − shared− −enable−linker−build−id− −libexecdir = /usr/lib− −without−included−gettext− −enable− threads = posix − −libdir = /usr/lib − −enable − nls − −with − sysroot = / − −enable − clocale = gnu− −enable−libstdcxx−debug− −enable−libstdcxx−time = yes− −with−default−libstdcxx−abi = new− −enable−gnu−unique−object− −disable−vtable−verify− −enable−libmpx− −enable−plugin− −enable − default − pie − −with − system − zlib − −with − target − system − zlib − −enable − objc − gc = auto − −enable − multiarch − −disable − werror − −with − arch − 32 = i686 − −with − abi = m64 − −with − multilib − list = m32, m64, mx32 − −enable − multilib − −with − tune = generic − −enable − offload − targets = nvptx − none − −without − cuda − driver − −enable − checking = release − −build = x86$_6$4 − linux − gnu − −host = x86$_6$4 − linux − gnu − −target = x86$_6$4 − linux − gnuThreadmodel : posixgccversion7.4.0(Ubuntu7.4.0 − 1ubuntu1 18.04.1)

## 10.2 SAT implementation

In this section I introduce the code of the implementation of the SAT algorithm, used to validate and qualify the FMB algorithm.

### 10.2.1 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 moving 2D Frame 'that' and 2D
// Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2DTime(
  const Frame2DTime* const that,
  const Frame2DTime* 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);

// Test for intersection between moving 3D Frame 'that' and 3D
// Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho);

#endif
```

### 10.2.2 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);

// Check the intersection constraint along one axis
bool CheckAxis3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho,
  const double* const axis,
  const double* const relSpeed);

// ------------ 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 iFrame = 2;
       iFrame--;) {
```

```
// 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 moving 2D Frame 'that' and 2D
// Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2DTime(
  const Frame2DTime* const that,
  const Frame2DTime* const tho) {

  // Declare a variable to loop on Frames and commonalize code
  const Frame2DTime* frameEdge = that;

  // Declare a variable to memorize the speed of tho relative to that
  double relSpeed[2];
  relSpeed[0] = tho->speed[0] - that->speed[0];
  relSpeed[1] = tho->speed[1] - that->speed[1];

  // Loop to commonalize code when checking SAT based on that's edges
  // and then tho's edges
  for (int iFrame = 2;
       iFrame--;) {

    // 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;

    }
```

```
// If the current frame is the second frame
if (iFrame == 1) {

  // Add one more edge to take into account the movement
  // of tho relative to that
  ++nbEdges;

}

// Loop on the frame's edges
for (int iEdge = nbEdges;
     iEdge--;) {

  // Get the current edge
  const double* edge =
    (iEdge == 3 ? relSpeed :
      (iEdge == 2 ?
        (frameEdgeType == FrameTetrahedron ? thirdEdge : relSpeed) :
        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 Frame2DTime* 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;

  }

  // If we are checking the second frame's vertices
  if (frame == tho) {

    // Check also the vertices moved by the relative speed
    vertex[0] += relSpeed[0];
    vertex[1] += relSpeed[1];

    proj = vertex[0] * edge[1] - vertex[1] * edge[0];

    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 iFrame = 2;
     iFrame--;) {

  // 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;

}

// Test for intersection between moving 3D Frame 'that' and 3D
// Frame 'tho'
```

```
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho) {

  // Declare two variables to memorize the opposite edges in case
  // of tetrahedron
  double oppEdgesThat[3][3];
  double oppEdgesTho[3][3];

  // Declare a variable to memorize the speed of tho relative to that
  double relSpeed[3];
  relSpeed[0] = tho->speed[0] - that->speed[0];
  relSpeed[1] = tho->speed[1] - that->speed[1];
  relSpeed[2] = tho->speed[2] - that->speed[2];

  // 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 Frame3DTime* 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 iFrame = 2;
       iFrame--;) {

    // 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[10][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;

}

// If we are checking the frame 'tho'
if (frame == tho) {

  // Add the normal to the virtual faces created by the speed
  // of tho relative to that

  normFaces[nbFaces][0] =
    relSpeed[1] * frameCompA[2] -
    relSpeed[2] * frameCompA[1];
  normFaces[nbFaces][1] =
    relSpeed[2] * frameCompA[0] -
    relSpeed[0] * frameCompA[2];
  normFaces[nbFaces][2] =
    relSpeed[0] * frameCompA[1] -
    relSpeed[1] * frameCompA[0];
  if (fabs(normFaces[nbFaces][0]) > EPSILON ||
      fabs(normFaces[nbFaces][1]) > EPSILON ||
      fabs(normFaces[nbFaces][2]) > EPSILON)
    ++nbFaces;

  normFaces[nbFaces][0] =
    relSpeed[1] * frameCompB[2] -
    relSpeed[2] * frameCompB[1];
  normFaces[nbFaces][1] =
    relSpeed[2] * frameCompB[0] -
    relSpeed[0] * frameCompB[2];
  normFaces[nbFaces][2] =
    relSpeed[0] * frameCompB[1] -
    relSpeed[1] * frameCompB[0];
  if (fabs(normFaces[nbFaces][0]) > EPSILON ||
      fabs(normFaces[nbFaces][1]) > EPSILON ||
      fabs(normFaces[nbFaces][2]) > EPSILON)
    ++nbFaces;

  normFaces[nbFaces][0] =
    relSpeed[1] * frameCompC[2] -
    relSpeed[2] * frameCompC[1];
  normFaces[nbFaces][1] =
```

```
    relSpeed[2] * frameCompC[0] -
    relSpeed[0] * frameCompC[2];
  normFaces[nbFaces][2] =
    relSpeed[0] * frameCompC[1] -
    relSpeed[1] * frameCompC[0];
  if (fabs(normFaces[nbFaces][0]) > EPSILON ||
      fabs(normFaces[nbFaces][1]) > EPSILON ||
      fabs(normFaces[nbFaces][2]) > EPSILON)
    ++nbFaces;

  if (frameType == FrameTetrahedron) {

    const double* oppEdgeA = oppEdgesA[0];
    const double* oppEdgeB = oppEdgesA[1];
    const double* oppEdgeC = oppEdgesA[2];

    normFaces[nbFaces][0] =
      relSpeed[1] * oppEdgeA[2] -
      relSpeed[2] * oppEdgeA[1];
    normFaces[nbFaces][1] =
      relSpeed[2] * oppEdgeA[0] -
      relSpeed[0] * oppEdgeA[2];
    normFaces[nbFaces][2] =
      relSpeed[0] * oppEdgeA[1] -
      relSpeed[1] * oppEdgeA[0];
    if (fabs(normFaces[nbFaces][0]) > EPSILON ||
        fabs(normFaces[nbFaces][1]) > EPSILON ||
        fabs(normFaces[nbFaces][2]) > EPSILON)
      ++nbFaces;

    normFaces[nbFaces][0] =
      relSpeed[1] * oppEdgeB[2] -
      relSpeed[2] * oppEdgeB[1];
    normFaces[nbFaces][1] =
      relSpeed[2] * oppEdgeB[0] -
      relSpeed[0] * oppEdgeB[2];
    normFaces[nbFaces][2] =
      relSpeed[0] * oppEdgeB[1] -
      relSpeed[1] * oppEdgeB[0];
    if (fabs(normFaces[nbFaces][0]) > EPSILON ||
        fabs(normFaces[nbFaces][1]) > EPSILON ||
        fabs(normFaces[nbFaces][2]) > EPSILON)
      ++nbFaces;

    normFaces[nbFaces][0] =
      relSpeed[1] * oppEdgeC[2] -
      relSpeed[2] * oppEdgeC[1];
    normFaces[nbFaces][1] =
      relSpeed[2] * oppEdgeC[0] -
      relSpeed[0] * oppEdgeC[2];
    normFaces[nbFaces][2] =
      relSpeed[0] * oppEdgeC[1] -
      relSpeed[1] * oppEdgeC[0];
    if (fabs(normFaces[nbFaces][0]) > EPSILON ||
        fabs(normFaces[nbFaces][1]) > EPSILON ||
        fabs(normFaces[nbFaces][2]) > EPSILON)
      ++nbFaces;

  }
}

// Loop on the frame's faces
```

```
  for (int iFace = nbFaces;
       iFace--;) {

    // Check against the current face's normal
    bool isIntersection =
      CheckAxis3DTime(
        that,
        tho,
        normFaces[iFace],
        relSpeed);

    // 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 + 1;
       iEdgeTho--;) {

    // Get the second edge
    const double* edgeTho =
      (iEdgeTho == nbEdgesTho ?
        relSpeed :
        (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 =
      CheckAxis3DTime(
        that,
        tho,
        axis,
        relSpeed);
```

```
      // 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;

}

// Check the intersection constraint for Frames 'that' and 'tho'
// relatively to 'axis'
bool CheckAxis3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho,
  const double* const axis,
  const double* const relSpeed) {

  // 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 Frame3DTime* 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;

    }

    // If we are checking the second frame's vertices
    if (frame == tho) {

      // Check also the vertices moved by the relative speed
      vertex[0] += relSpeed[0];
      vertex[1] += relSpeed[1];
      vertex[2] += relSpeed[2];

    proj =
      vertex[0] * axis[0] +
      vertex[1] * axis[1] +
      vertex[2] * axis[2];

      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;

}
```

## 10.3 Makefile

In this section I introduce the Makefile used to compile the code given in the previous sections.

```
COMPILER=gcc
OPTIMIZATION=-O3

all : compile run plot doc

install :
	sudo apt-get install gnuplot

compile : main unitTests validation qualification

main : main2D main2DTime main3D main3DTime

main2D:
	cd 2D; make main; cd -

main2DTime:
	cd 2DTime; make main; cd -

main3D:
	cd 3D; make main; cd -

main3DTime:
	cd 3DTime; make main; cd -

unitTests : unitTests2D unitTests2DTime unitTests3D unitTests3DTime

unitTests2D:
	cd 2D; make unitTests; cd -

unitTests2DTime:
	cd 2DTime; make unitTests; cd -

unitTests3D:
	cd 3D; make unitTests; cd -

unitTests3DTime:
	cd 3DTime; make unitTests; cd -

validation : validation2D validation2DTime validation3D validation3DTime

validation2D:
	cd 2D; make validation; cd -
```

```
validation2DTime:
	cd 2DTime; make validation; cd -

validation3D:
	cd 3D; make validation; cd -

validation3DTime:
	cd 3DTime; make validation; cd -

qualification : qualification2D qualification2DTime qualification3D
    qualification3DTime

qualification2D:
	cd 2D; make qualification; cd -

qualification2DTime:
	cd 2DTime; make qualification; cd -

qualification3D:
	cd 3D; make qualification; cd -

qualification3DTime:
	cd 3DTime; make qualification; cd -

clean : clean2D clean2DTime clean3D clean3DTime

clean2D:
	cd 2D; make clean; cd -

clean2DTime:
	cd 2DTime; make clean; cd -

clean3D:
	cd 3D; make clean; cd -

clean3DTime:
	cd 3DTime; make clean; cd -

valgrind : valgrind2D valgrind2DTime valgrind3D valgrind3DTime

valgrind2D:
	cd 2D; make valgrind; cd -

valgrind2DTime:
	cd 2DTime; make valgrind; cd -

valgrind3D:
	cd 3D; make valgrind; cd -

valgrind3DTime:
	cd 3DTime; make valgrind; cd -

run : run2D run2DTime run3D run3DTime

run2D:
	cd 2D; ./main > ../Results/main2D.txt; ./unitTests > ../Results/
	    unitTests2D.txt; ./validation > ../Results/validation2D.txt;
	    grep failed ../Results/validation2D.txt; ./qualification > ../
	    Results/qualification2D.txt; grep failed ../Results/
	    qualification2D.txt; cd -

run3D:
```

```
        cd 3D; ./main > ../Results/main3D.txt; ./unitTests > ../Results/
            unitTests3D.txt; ./validation > ../Results/validation3D.txt;
            grep failed ../Results/validation3D.txt; ./qualification > ../
            Results/qualification3D.txt; grep failed ../Results/
            qualification3D.txt; cd -

run2DTime:
        cd 2DTime; ./main > ../Results/main2DTime.txt; ./unitTests > ../
            Results/unitTests2DTime.txt; ./validation > ../Results/
            validation2DTime.txt; grep failed ../Results/validation2DTime.
            txt; ./qualification > ../Results/qualification2DTime.txt; grep
            failed ../Results/qualification2DTime.txt; cd -

run3DTime:
        cd 3DTime; ./main > ../Results/main3DTime.txt; ./unitTests > ../
            Results/unitTests3DTime.txt; ./validation > ../Results/
            validation3DTime.txt; grep failed ../Results/validation3DTime.
            txt; ./qualification > ../Results/qualification3DTime.txt; grep
            failed ../Results/qualification3DTime.txt; cd -

plot: cleanPlot plot2D plot2DTime plot3D plot3DTime

cleanPlot:
        rm Results/*.png

plot2D:
        cd Results; gnuplot qualification2D.gnu < qualification2D.txt; cd -

plot2DTime:
        cd Results; gnuplot qualification2DTime.gnu < qualification2DTime.
            txt; cd -

plot3D:
        cd Results; gnuplot qualification3D.gnu < qualification3D.txt; cd -

plot3DTime:
        cd Results; gnuplot qualification3DTime.gnu < qualification3DTime.
            txt; cd -

doc:
        cd Doc; make latex; cd -
```

### 10.3.1 2D static

```
all : main unitTests validation qualification

COMPILER?=gcc
OPTIMIZATION?=-O3
BUILD_ARG=$(OPTIMIZATION) -I../SAT -I../Frame

main : main.o fmb2d.o frame.o Makefile
        $(COMPILER) -o main main.o fmb2d.o frame.o

main.o : main.c fmb2d.h ../Frame/frame.h Makefile
        $(COMPILER) -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb2d.o frame.o Makefile
        $(COMPILER) -o unitTests unitTests.o fmb2d.o frame.o $(LINK_ARG)
```

```
unitTests.o : unitTests.c fmb2d.h ../Frame/frame.h Makefile
	$(COMPILER) -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb2d.o sat.o frame.o Makefile
	$(COMPILER) -o validation validation.o fmb2d.o sat.o frame.o

validation.o : validation.c fmb2d.h ../SAT/sat.h ../Frame/frame.h Makefile
	$(COMPILER) -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb2d.o sat.o frame.o Makefile
	$(COMPILER) -o qualification qualification.o fmb2d.o sat.o frame.o $
	    (LINK_ARG)

qualification.o : qualification.c fmb2d.h ../SAT/sat.h ../Frame/frame.h
    Makefile
	$(COMPILER) -c qualification.c $(BUILD_ARG)

fmb2d.o : fmb2d.c fmb2d.h ../Frame/frame.h Makefile
	$(COMPILER) -c fmb2d.c $(BUILD_ARG)

sat.o : ../SAT/sat.c ../SAT/sat.h ../Frame/frame.h Makefile
	$(COMPILER) -c ../SAT/sat.c $(BUILD_ARG)

frame.o : ../Frame/frame.c ../Frame/frame.h Makefile
	$(COMPILER) -c ../Frame/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
```

### 10.3.2 3D static

```
all : main unitTests validation qualification

COMPILER?=gcc
OPTIMIZATION?=-O3
BUILD_ARG=$(OPTIMIZATION) -I../SAT -I../Frame

main : main.o fmb3d.o frame.o Makefile
	$(COMPILER) -o main main.o fmb3d.o frame.o

main.o : main.c fmb3d.h ../Frame/frame.h Makefile
	$(COMPILER) -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb3d.o frame.o Makefile
	$(COMPILER) -o unitTests unitTests.o fmb3d.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb3d.h ../Frame/frame.h Makefile
	$(COMPILER) -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb3d.o sat.o frame.o Makefile
	$(COMPILER) -o validation validation.o fmb3d.o sat.o frame.o

validation.o : validation.c fmb3d.h ../SAT/sat.h ../Frame/frame.h Makefile
	$(COMPILER) -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb3d.o sat.o frame.o Makefile
	$(COMPILER) -o qualification qualification.o fmb3d.o sat.o frame.o $
```

```
            (LINK_ARG)

qualification.o : qualification.c fmb3d.h ../SAT/sat.h ../Frame/frame.h
    Makefile
        $(COMPILER) -c qualification.c $(BUILD_ARG)

fmb3d.o : fmb3d.c fmb3d.h ../Frame/frame.h Makefile
        $(COMPILER) -c fmb3d.c $(BUILD_ARG)

sat.o : ../SAT/sat.c ../SAT/sat.h ../Frame/frame.h Makefile
        $(COMPILER) -c ../SAT/sat.c $(BUILD_ARG)

frame.o : ../Frame/frame.c ../Frame/frame.h Makefile
        $(COMPILER) -c ../Frame/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
```

### 10.3.3 2D dynamic

```
all : main unitTests validation qualification

COMPILER?=gcc
OPTIMIZATION?=-O3
BUILD_ARG=$(OPTIMIZATION) -I../SAT -I../Frame

main : main.o fmb2dt.o frame.o Makefile
        $(COMPILER) -o main main.o fmb2dt.o frame.o

main.o : main.c fmb2dt.h ../Frame/frame.h Makefile
        $(COMPILER) -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb2dt.o frame.o Makefile
        $(COMPILER) -o unitTests unitTests.o fmb2dt.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb2dt.h ../Frame/frame.h Makefile
        $(COMPILER) -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb2dt.o sat.o frame.o Makefile
        $(COMPILER) -o validation validation.o fmb2dt.o sat.o frame.o

validation.o : validation.c fmb2dt.h ../SAT/sat.h ../Frame/frame.h Makefile
        $(COMPILER) -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb2dt.o sat.o frame.o Makefile
        $(COMPILER) -o qualification qualification.o fmb2dt.o sat.o frame.o
            $(LINK_ARG)

qualification.o : qualification.c fmb2dt.h ../SAT/sat.h ../Frame/frame.h
    Makefile
        $(COMPILER) -c qualification.c $(BUILD_ARG)

fmb2dt.o : fmb2dt.c fmb2dt.h ../Frame/frame.h Makefile
        $(COMPILER) -c fmb2dt.c $(BUILD_ARG)

sat.o : ../SAT/sat.c ../SAT/sat.h ../Frame/frame.h Makefile
        $(COMPILER) -c ../SAT/sat.c $(BUILD_ARG)
```

```
frame.o : ../Frame/frame.c ../Frame/frame.h Makefile
	$(COMPILER) -c ../Frame/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
```

### 10.3.4 3D dynamic

```
all : main unitTests validation qualification

COMPILER?=gcc
OPTIMIZATION?=-O3
BUILD_ARG=$(OPTIMIZATION) -I../SAT -I../Frame

main : main.o fmb3dt.o frame.o Makefile
	$(COMPILER) -o main main.o fmb3dt.o frame.o

main.o : main.c fmb3dt.h ../Frame/frame.h Makefile
	$(COMPILER) -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb3dt.o frame.o Makefile
	$(COMPILER) -o unitTests unitTests.o fmb3dt.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb3dt.h ../Frame/frame.h Makefile
	$(COMPILER) -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb3dt.o sat.o frame.o Makefile
	$(COMPILER) -o validation validation.o fmb3dt.o sat.o frame.o

validation.o : validation.c fmb3dt.h ../SAT/sat.h ../Frame/frame.h Makefile
	$(COMPILER) -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb3dt.o sat.o frame.o Makefile
	$(COMPILER) -o qualification qualification.o fmb3dt.o sat.o frame.o
	    $(LINK_ARG)

qualification.o : qualification.c fmb3dt.h ../SAT/sat.h ../Frame/frame.h
    Makefile
	$(COMPILER) -c qualification.c $(BUILD_ARG)

fmb3dt.o : fmb3dt.c fmb3dt.h ../Frame/frame.h Makefile
	$(COMPILER) -c fmb3dt.c $(BUILD_ARG)

sat.o : ../SAT/sat.c ../SAT/sat.h ../Frame/frame.h Makefile
	$(COMPILER) -c ../SAT/sat.c $(BUILD_ARG)

frame.o : ../Frame/frame.c ../Frame/frame.h Makefile
	$(COMPILER) -c ../Frame/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.