¡¡¡¡¡¡¡ HEAD ======= ¿¿¿¿¿¿¿ test$_{o}ptim <<<<<<< HEAD =======>>>>>>> test_{o}ptim <<<<<<< HEAD ======= >>>>>>> test_{o}ptim$

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 three following sections.

The last three 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.

Let's call $\mathbb{A}$ and $\mathbb{B}$ the two Frames tested for intersection. If $\mathbb{A}$ and $\mathbb{B}$ 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 $\mathbb{A}$ (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, $\mathbb{A}_{\mathbb{A}} \cap \mathbb{B}_{\mathbb{A}}$,can then be expressed as follow.

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

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

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

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

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

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

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

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

These can in turn be expressed as systems of linear inequations as follow, 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:

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

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

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

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

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

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

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

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 $\mathbb{A}$ and $\mathbb{B}$ 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 $\mathbb{A}$ (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, $\mathbb{B}_{\mathbb{A}}$ becomes:

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

$\mathbb{A}$ 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, $\mathbb{A}_{\mathbb{A}} \cap \mathbb{B}_{\mathbb{A}}$, can then be expressed as follow.

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

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

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

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

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

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

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

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

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:

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

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

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

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

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

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

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

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

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

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

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

where

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

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 allows 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 the inequality systems of the previous section, to obtain the bounding box of the intersection, if the system has a solution. If the system has no solution, the method will eventually reach an inconsistent inequality, meaning there is no intersection between the two Frames.

One coordinate $\overrightarrow{S}$, or $(\overrightarrow{S}, t)$ in dynamic case, 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 $\overrightarrow{S'}$, or $(\overrightarrow{S}, t)$, in the real world's coordinates system as follow:

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

$$(\overrightarrow{S'}, t) = \left(\overrightarrow{O_{\mathbb{B}}} + C_{\mathbb{B}}.\overrightarrow{S} + \overrightarrow{V}_{\mathbb{B}}.t, t\right) \tag{40}$$

Only one inconsistent inequality is sufficient to prove the absence of solution, and then the non intersection of the Frames. Thus, 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 sufficient condition for one inequality $\sum_i a_i X_i \leq Y$ to be inconsistent is, given that $\forall i, X_i \in [0.0, 1.0]$:

$$Y < \sum_{i \in I^-} a_i \tag{41}$$

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

2.3 About the size of the system of linear inequations

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.

Let's call n_-, n_+ and n_0, each in $[0, \mathbb{N}]$, 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:

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

and

$$n_-.n_+ + n_0 = N' \tag{43}$$

Now let's define $K = N - n_0$, then we have:

$$n_- + n_+ = K \tag{44}$$

then,

$$n_-.n_+ = n_-.(K - n_-) \tag{45}$$

then,

$$n_-.n_+ = K.n_- - n_-^2 \tag{46}$$

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

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

or,

$$n_-.n_+ \leq K^2/4 \tag{48}$$

and putting back the definition of K

$$n_-.n_+ \leq (N - n_0)^2/4 \tag{49}$$

which is also

$$n_-.n_+ \leq N^2/4 \tag{50}$$

From (43) we get,

$$N' \leq N^2/4 + n_0 \tag{51}$$

and finally,

$$N' \leq N^2/4 + N \tag{52}$$

The maximum number of inequations 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	24		
$2Ddynamic$	10	35	342	
$3Dstatic$	12	48	624	
$3Ddynamic$	14	63	1056	279840

However, these theoretical values are much higher than the ones encountered in practice, and the maximum number of inequations encountered during validation were:

	N	N'	N''	N'''
$2Dstatic$	8	11		
$2Ddynamic$	10	13	21	
$3Dstatic$	12	20	55	
$3Ddynamic$	14	22	57	560

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 Frames.

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 section for possible optimization in this language.

Algorithms are also given independantly from each other. Code commonalization may be possible if one plans to use 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 TO (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 TO 1
    PRINT that.orig[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  comp = ["x","y"]
  FOR j = 0 TO 1
    PRINT ") ", comp[j], "("
    FOR i = 0 TO 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 TO 1
    PRINT that.min[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  PRINT ")-maxXY("
  FOR i = 0 TO 1
    PRINT that.max[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  PRINT ")"
```

```
END FUNCTION

FUNCTION Frame2DExportBdgBox(that, bdgBox, bdgBoxProj)
  FOR i = 0 TO 1
    bdgBoxProj.max[i] = that.orig[i]
    FOR j = 0 TO 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 TO (nbVertices - 1)
    FOR i = 0 TO 1
      IF BITWISEAND(iVertex, powi(2, i)) <> 0
        v[i] = bdgBox.max[i]
      ELSE
        v[i] = bdgBox.min[i]
      END IF
    END FOR
    FOR i = 0 TO 1
      w[i] = that.orig[i]
      FOR j = 0 TO 1
        w[i] = w[i] + that.comp[j][i] * v[j]
      END FOR
    END FOR
    FOR i = 0 TO 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 TO 1
    v[i] = Q.orig[i] - P.orig[i]
  END FOR
  FOR i = 0 TO 1
    Qp.orig[i] = 0.0
    FOR j = 0 TO 1
      Qp.orig[i] = Qp.orig[i] + P.invComp[j][i] * v[j]
      Qp.comp[j][i] = 0.0
      FOR k = 0 TO 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 TO 1
    that.orig[iAxis] = orig[iAxis]
    FOR iComp = 0 TO 1
      that.comp[iComp][iAxis] = comp[iComp][iAxis]
    END FOR
  END FOR
  FOR iAxis = 0 TO 1
    min = orig[iAxis]
    max = orig[iAxis]
    FOR iComp = 0 TO 1
      IF that.type == FrameCuboid
        IF that.comp[iComp][iAxis] < 0.0
          min = min + that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0
          max = 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 TO (nbRows - 2)
    IF M[iRow][iVar] <> 0.0
      FOR jRow = (iRow + 1) TO (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 TO (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 TO (nbRows - 1)
    IF M[iRow][iVar] == 0.0
      jCol = 0
      FOR iCol = 0 TO (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 TO (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
  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 IF
  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 IF
  GetBound2D(SND_VAR, Mp, Yp, nbRowsP, bdgBoxLocal)
  IF bdgBoxLocal.min[SND_VAR] >= bdgBoxLocal.max[SND_VAR]
    RETURN FALSE
  END IF
  ElimVar2D(SND_VAR, M, Y, nbRows, 2, Mp, Yp, nbRowsP)
  GetBound2D(FST_VAR, Mp, Yp, nbRowsP, bdgBoxLocal)
  bdgBox = bdgBoxLocal
  RETURN TRUE
END FUNCTION

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 TO (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 TO 2
    PRINT that.orig[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  comp = ["x","y","z"]
  FOR j = 0 TO 2
    PRINT ") ", comp[j], "("
    FOR i = 0 TO 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 TO 2
    PRINT that.min[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  PRINT ")-maxXYZ("
  FOR i = 0 TO 2
    PRINT that.max[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  PRINT ")"
END FUNCTION

FUNCTION Frame3DExportBdgBox(that, bdgBox, bdgBoxProj)
  FOR i = 0 TO 2
    bdgBoxProj.max[i] = that.orig[i]
    FOR j = 0 TO 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 TO (nbVertices - 1)
    FOR i = 0 TO 2
      IF BITWISEAND(iVertex, powi(2, i)) <> 0
        v[i] = bdgBox.max[i]
      ELSE
```

```
        v[i] = bdgBox.min[i]
      END IF
    END FOR
    FOR i = 0 TO 2
      w[i] = that.orig[i]
      FOR j = 0 TO 2
        w[i] = w[i] + that.comp[j][i] * v[j]
      END FOR
    END FOR
    FOR i = 0 TO 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 TO 2
    v[i] = Q.orig[i] - P.orig[i]
  END FOR
  FOR i = 0 TO 2
    Qp.orig[i] = 0.0
    FOR j = 0 TO 2
      Qp.orig[i] = Qp.orig[i] + P.invComp[j][i] * v[j]
      Qp.comp[j][i] = 0.0
      FOR k = 0 TO 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 TO 2
    that.orig[iAxis] = orig[iAxis]
    FOR iComp = 0 TO 2
      that.comp[iComp][iAxis] = comp[iComp][iAxis]
    END FOR
  END FOR
  FOR iAxis = 0 TO 2
    min = orig[iAxis]
    max = orig[iAxis]
    FOR iComp = 0 TO 2
      IF that.type == FrameCuboid
        IF that.comp[iComp][iAxis] < 0.0
          min = min + that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0
          max = 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 TO (nbRows - 2)
    IF M[iRow][iVar] <> 0.0
      FOR jRow = (iRow + 1) TO (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 TO (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 TO (nbRows - 1)
    IF M[iRow][iVar] == 0.0
      jCol = 0
      FOR iCol = 0 TO (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 TO (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
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 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] = 0.0
nbRows = nbRows + 1
inconsistency =
  ElimVar3D(FST_VAR, M, Y, nbRows, 3, Mp, Yp, nbRowsP)
IF inconsistency == TRUE
  RETURN FALSE
END IF
inconsistency =
  ElimVar3D(FST_VAR, Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
IF inconsistency == TRUE
  RETURN FALSE
END IF
GetBound3D(THD_VAR, Mpp, Ypp, nbRowsPP, bdgBoxLocal)
IF bdgBoxLocal.min[THD_VAR] >= bdgBoxLocal.max[THD_VAR]
  RETURN FALSE
END IF
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 FUNCTION

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 = [0.0, 0.0, 0.0]
compQ3D = [
  [1.0, 0.0, 0.0],
  [0.0, 1.0, 0.0],
  [0.0, 0.0, 1.0]]
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 TO (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 TO 1
    PRINT that.orig[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  PRINT ") s("
  FOR i = 0 TO 1
    PRINT that.speed[i]
    IF i < 1
      PRINT ","
    END IF
  END FOR
  comp = ["x", "y"]
  FOR j = 0 TO 1
    PRINT ") ", comp[j], "("
    FOR i = 0 TO 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 TO 2
    PRINT that.min[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  PRINT ")-maxXYT("
  FOR i = 0 TO 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 TO 1
    bdgBoxProj.max[i] = that.orig[i] + that.speed[i] * bdgBox.min[2]
    FOR j = 0 TO 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 TO (nbVertices - 1)
    FOR i = 0 TO 1
      IF BITWISEAND(iVertex, powi(2, i)) <> 0
        v[i] = bdgBox.max[i]
      ELSE
        v[i] = bdgBox.min[i]
      END IF
    END FOR
    FOR i = 0 TO 1
      w[i] = that.orig[i]
      FOR j = 0 TO 1
        w[i] = w[i] + that.comp[j][i] * v[j]
      END FOR
    END FOR
    FOR i = 0 TO 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 TO 1
    v[i] = Q.orig[i] - P.orig[i]
    s[i] = Q.speed[i] - P.speed[i]
  END FOR
  FOR i = 0 TO 1
    Qp.orig[i] = 0.0
    Qp.speed[i] = 0.0
    FOR j = 0 TO 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 TO 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 TO 1
```

```
    that.orig[iAxis] = orig[iAxis]
    that.speed[iAxis] = speed[iAxis]
    FOR iComp = 0 TO 1
      that.comp[iComp][iAxis] = comp[iComp][iAxis]
    END FOR
  END FOR
  FOR iAxis = 0 TO 1
    min = orig[iAxis]
    max = orig[iAxis]
    FOR iComp = 0 TO 1
      IF that.type == FrameCuboid
        IF that.comp[iComp][iAxis] < 0.0
          min = min + that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0
          max = 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 TO (nbRows - 2)
    IF M[iRow][iVar] <> 0.0
      FOR jRow = (iRow + 1) TO (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 TO (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 TO (nbRows - 1)
    IF M[iRow][iVar] == 0.0
      jCol = 0
      FOR iCol = 0 TO (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 TO (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 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
    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]
    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
    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
    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 TO (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 TO 2
    PRINT that.orig[i]
    IF i < 2
      PRINT ","
    END IF
```

```
  END FOR
  PRINT " s("
  FOR i = 0 TO 2
    PRINT that.speed[i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
  comp = ["x", "y", "z"]
  FOR j = 0 TO 2
    PRINT " ", comp[j], "("
    FOR i = 0 TO 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 TO 3
    PRINT that.min[i]
    IF i < 3
      PRINT ","
    END IF
  END FOR
  PRINT ")-maxXYZT("
  FOR i = 0 TO 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 TO 2
    bdgBoxProj.max[i] = that.orig[i] + that.speed[i] * bdgBox.min[3]
    FOR j = 0 TO 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 TO (nbVertices - 1)
    FOR i = 0 TO 2
      IF BITWISEAND(iVertex, powi(2, i)) <> 0
        v[i] = bdgBox.max[i]
      ELSE
        v[i] = bdgBox.min[i]
      END IF
    END FOR
    FOR i = 0 TO 2
      w[i] = that.orig[i]
      FOR j = 0 TO 2
```

```
        w[i] = w[i] + that.comp[j][i] * v[j]
      END FOR
    END FOR
    FOR i = 0 TO 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 TO 2
    v[i] = Q.orig[i] - P.orig[i]
    s[i] = Q.speed[i] - P.speed[i]
  END FOR
  FOR i = 0 TO 2
    Qp.orig[i] = 0.0
    Qp.speed[i] = 0.0
    FOR j = 0 TO 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 TO 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 TO 2
    that.orig[iAxis] = orig[iAxis]
    that.speed[iAxis] = speed[iAxis]
    FOR iComp = 0 TO 2
      that.comp[iComp][iAxis] = comp[iComp][iAxis]
    END FOR
  END FOR
  FOR iAxis = 0 TO 2
    min = orig[iAxis]
    max = orig[iAxis]
    FOR iComp = 0 TO 2
      IF that.type == FrameCuboid
        IF that.comp[iComp][iAxis] < 0.0
          min = min + that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0
          max = 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 TO (nbRows - 2)
    IF M[iRow][iVar] <> 0.0
      FOR jRow = (iRow + 1) TO (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 TO (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 TO (nbRows - 1)
    IF M[iRow][iVar] == 0.0
      jCol = 0
      FOR iCol = 0 TO (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 TO (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
  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]
    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 == TRUE
  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
// T/C  <- type of Frame
// o(orig[0], orig[1], orig[2])
// s(speed[0], speed[1], speed[2])
// x(comp[0][0], comp[0][1], comp[0][2])
// y(comp[1][0], comp[1][1], comp[1][2])
// z(comp[2][0], comp[2][1], comp[2][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])
// and 'speed'
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 'bdgBoxProj' 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' and 'speed'
// 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 'bdgBoxProj' 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
// T/C  <- type of Frame
// o(orig[0], orig[1], orig[2])
// s(speed[0], speed[1], speed[2])
// x(comp[0][0], comp[0][1], comp[0][2])
// y(comp[1][0], comp[1][1], comp[1][2])
// z(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 their 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) {

  // 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 tho
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 their 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 tho
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 inconsistent
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 tho
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 inconsistent
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 memorize 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 inconsistent
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 for each case 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>

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

// Epslon for numerical precision
#define EPSILON 0.0001

// Helper structure to pass arguments to the UnitTest function
typedef struct {
  FrameType type;
  double orig[2];
  double comp[2][2];
} Param2D;

// Unit test function
// Takes two Frame definitions, the correct answer in term of
// intersection/no intersection and the correct bounding box
// Run the FMB intersection detection alogirhtm on the Frames
// and check against the correct results
void UnitTest2D(
        const Param2D paramP,
        const Param2D paramQ,
         const bool correctAnswer,
  const AABB2D* const correctBdgBox) {

  // Create the two Frames
  Frame2D P =
    Frame2DCreateStatic(
      paramP.type,
```

```
    paramP.orig,
    paramP.comp);

Frame2D Q =
  Frame2DCreateStatic(
    paramQ.type,
    paramQ.orig,
    paramQ.comp);

// Declare a variable to memorize the resulting bounding box
AABB2D bdgBoxLocal;

// Helper variables to loop on the pair (that, tho) and (tho, that)
Frame2D* that = &P;
Frame2D* tho = &Q;

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

  // Display the tested frames
  Frame2DPrint(that);
  printf("\nagainst\n");
  Frame2DPrint(tho);
  printf("\n");

  // Run the FMB intersection test
  bool isIntersecting =
    FMBTestIntersection2D(
      that,
      tho,
      &bdgBoxLocal);

  // If the test hasn't given the expected answer about intersection
  if (isIntersecting != correctAnswer) {

    // Display information about the failure
    printf(" Failed\n");
    printf("Expected : ");
    if (correctAnswer == false)
      printf("no ");
    printf("intersection\n");
    printf("Got : ");
    if (isIntersecting == false)
      printf("no ");
    printf("intersection\n");
    exit(0);

  // Else, the test has given the expected answer about intersection
  } else {

    // If the Frames were intersecting
    if (isIntersecting == true) {

      AABB2D bdgBox;
      Frame2DExportBdgBox(
        tho,
        &bdgBoxLocal,
        &bdgBox);

      for (int iAxis = 2;
           iAxis--;) {
```

```
      if (bdgBox.min[iAxis] < that->bdgBox.min[iAxis]) {
        bdgBox.min[iAxis] = that->bdgBox.min[iAxis];
      }
      if (bdgBox.max[iAxis] > that->bdgBox.max[iAxis]) {
        bdgBox.max[iAxis] = that->bdgBox.max[iAxis];
      }
      if (bdgBox.min[iAxis] < tho->bdgBox.min[iAxis]) {
        bdgBox.min[iAxis] = tho->bdgBox.min[iAxis];
      }
      if (bdgBox.max[iAxis] > tho->bdgBox.max[iAxis]) {
        bdgBox.max[iAxis] = tho->bdgBox.max[iAxis];
      }
    }

    // Check the bounding box
    bool flag = true;
    for (int i = 2;
         i--;) {

      if (bdgBox.min[i] > correctBdgBox->min[i] + EPSILON ||
          bdgBox.max[i] < correctBdgBox->max[i] - EPSILON) {

        flag = false;

      }

    }

    // If the bounding box is the expected one
    if (flag == true) {

      // Display information
      printf("Succeed\n");

    // Else, the bounding box wasn't the expected one
    } else {

      // Display information
      printf("Failed\n");
      printf("Expected : ");
      AABB2DPrint(correctBdgBox);
      printf("\n");
      printf("     Got : ");
      AABB2DPrint(&bdgBox);
      printf("\n");

      // Terminate the unit tests
      exit(0);

    }

  // Else the Frames were not intersected,
  // no need to check the bounding box
  } else {

    // Display information
    printf(" Succeed\n");

  }

}
```

```
    printf("\n");

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

  }

}

void Test2D(void) {

  // Declare two variables to memozie the arguments to the
  // Validation function
  Param2D paramP;
  Param2D paramQ;

  // Declare a variable to memorize the correct bounding box
  AABB2D correctBdgBox;

  // Execute the unit test on various cases

  // ----------------------------
  paramP = (Param2D)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0},
     .comp =
       {{1.0, 0.0},
        {0.0, 1.0}}
    };
  paramQ = (Param2D)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0},
     .comp =
       {{1.0, 0.0},
        {0.0, 1.0}}
    };
  correctBdgBox = (AABB2D)
    {.min = {0.0, 0.0},
     .max = {1.0, 1.0}
    };
  UnitTest2D(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

  // ----------------------------
  paramP = (Param2D)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0},
     .comp =
       {{1.0, 0.0},
        {0.0, 1.0}}
    };
  paramQ = (Param2D)
    {.type = FrameCuboid,
     .orig = {0.5, 0.5},
     .comp =
       {{1.0, 0.0},
        {0.0, 1.0}}
    };
```

```
correctBdgBox = (AABB2D)
  {.min = {0.5, 0.5},
   .max = {1.0, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {-0.5, -0.5},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.5, 0.5},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
UnitTest2D(
  paramP,
  paramQ,
  false,
  NULL);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.25, -0.25},
   .comp =
     {{0.5, 0.0},
      {0.0, 2.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.25, 0.0},
   .max = {0.75, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
```

```
      {0.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {-0.25, 0.25},
   .comp =
     {{2.0, 0.0},
      {0.0, 0.5}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.0, 0.25},
   .max = {1.0, 0.75}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 1.0},
      {-1.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.0, 0.0},
   .max = {1.0, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {-0.5, -0.5},
   .comp =
     {{1.0, 1.0},
      {-1.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.0, 0.0},
   .max = {0.5, 1.0}
```

```
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {1.5, 1.5},
   .comp =
     {{1.0, -1.0},
      {-1.0, -1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {1.0, 0.0},
   .comp =
     {{-1.0, 0.0},
      {0.0, 1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.5, 0.0},
   .max = {1.0, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {1.0, 0.5},
   .comp =
     {{-0.5, 0.5},
      {-0.5, -0.5}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 1.0},
   .comp =
     {{1.0, 0.0},
      {0.0, -1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.0, 0.0},
   .max = {1.0, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
```

```
    {{1.0, 0.0},
     {1.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {2.0, -1.0},
   .comp =
     {{0.0, 1.0},
      {-0.5, 1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {1.5, 0.5},
   .max = {1.5 + 0.5 / 3.0, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.5},
      {0.5, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {1.0, 1.0},
   .comp =
     {{-0.5, -0.5},
      {0.0, -1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.5, 0.25},
   .max = {1.0, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.5},
      {0.5, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {1.0, 2.0},
   .comp =
     {{-0.5, -0.5},
      {0.0, -1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.5, 0.75},
```

```
    .max = {1.0, 1.25}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.5},
      {0.5, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameCuboid,
   .orig = {1.0, 2.0},
   .comp =
     {{-0.5, -0.5},
      {0.0, -1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.5, 0.5},
   .max = {0.75, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.5},
      {0.5, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {1.0, 2.0},
   .comp =
     {{-0.5, -0.5},
      {0.0, -1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.5 + 1.0 / 3.0, 1.0},
   .max = {1.0, 1.0 + 1.0 / 3.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
```

```
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.0, 0.0},
   .max = {1.0, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {0.0, -0.5},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.0, 0.0},
   .max = {0.5, 0.5}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.5, 0.5},
   .comp =
     {{-0.5, 0.0},
      {0.0, -0.5}}
  };
paramQ = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {0.0, -0.5},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
correctBdgBox = (AABB2D)
```

```
  {.min = {0.0, 0.0},
   .max = {0.5, 0.5}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.5, 0.5},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
UnitTest2D(
  paramP,
  paramQ,
  false,
  NULL);

// ---------------------------
paramP = (Param2D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {1.5, 1.5},
   .comp =
     {{-1.5, 0.0},
      {0.0, -1.5}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.5, 0.5},
   .max = {1.0, 1.0}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
```

```
  };
paramQ = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {1.01, 1.01},
   .comp =
     {{-1.0, 0.0},
      {0.0, -1.0}}
  };
UnitTest2D(
  paramP,
  paramQ,
  false,
  NULL);

// ---------------------------
paramP = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.5},
      {0.5, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {1.0, 1.0},
   .comp =
     {{-0.5, -0.5},
      {0.0, -1.0}}
  };
correctBdgBox = (AABB2D)
  {.min = {0.5, 0.5 - 1.0 / 6.0},
   .max = {1.0, 0.75}
  };
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.5},
      {0.5, 1.0}}
  };
paramQ = (Param2D)
  {.type = FrameTetrahedron,
   .orig = {1.01, 1.5},
   .comp =
     {{-0.5, -0.5},
      {0.0, -1.0}}
  };
UnitTest2D(
  paramP,
  paramQ,
  false,
  NULL);

// If we reached here, it means all the unit tests succeed
printf("All unit tests 2D have succeed.\n");
```

```
}

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

  Test2D();

  return 0;
}
```

6.1.2 3D static

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

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

// Epslon for numerical precision
#define EPSILON 0.0001

// Helper structure to pass arguments to the UnitTest function
typedef struct {
  FrameType type;
  double orig[3];
  double comp[3][3];
} Param3D;

// Unit test function
// Takes two Frame definitions, the correct answer in term of
// intersection/no intersection and the correct bounding box
// Run the FMB intersection detection alogirhtm on the Frames
// and check against the correct results
void UnitTest3D(
        const Param3D paramP,
        const Param3D paramQ,
         const bool correctAnswer,
  const AABB3D* const correctBdgBox) {

  // Create the two Frames
  Frame3D P =
    Frame3DCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.comp);

  Frame3D Q =
    Frame3DCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.comp);

  // Declare a variable to memorize the resulting bounding box
  AABB3D bdgBoxLocal;

  // Helper variables to loop on the pair (that, tho) and (tho, that)
  Frame3D* that = &P;
  Frame3D* tho = &Q;
```

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

  // Display the tested frames
  Frame3DPrint(that);
  printf("\nagainst\n");
  Frame3DPrint(tho);
  printf("\n");

  // Run the FMB intersection test
  bool isIntersecting =
    FMBTestIntersection3D(
      that,
      tho,
      &bdgBoxLocal);

  // If the test hasn't given the expected answer about intersection
  if (isIntersecting != correctAnswer) {

    // Display information about the failure
    printf(" Failed\n");
    printf("Expected : ");
    if (correctAnswer == false)
      printf("no ");
    printf("intersection\n");
    printf("Got : ");
    if (isIntersecting == false)
      printf("no ");
    printf("intersection\n");
    exit(0);

  // Else, the test has given the expected answer about intersection
  } else {

    // If the Frames were intersecting
    if (isIntersecting == true) {

      AABB3D bdgBox;
      Frame3DExportBdgBox(
        tho,
        &bdgBoxLocal,
        &bdgBox);

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

        if (bdgBox.min[iAxis] < that->bdgBox.min[iAxis]) {
          bdgBox.min[iAxis] = that->bdgBox.min[iAxis];
        }
        if (bdgBox.max[iAxis] > that->bdgBox.max[iAxis]) {
          bdgBox.max[iAxis] = that->bdgBox.max[iAxis];
        }
        if (bdgBox.min[iAxis] < tho->bdgBox.min[iAxis]) {
          bdgBox.min[iAxis] = tho->bdgBox.min[iAxis];
        }
        if (bdgBox.max[iAxis] > tho->bdgBox.max[iAxis]) {
          bdgBox.max[iAxis] = tho->bdgBox.max[iAxis];
        }
      }
```

```
        // Check the bounding box
        bool flag = true;
        for (int i = 3;
             i--;) {

          if (bdgBox.min[i] > correctBdgBox->min[i] + EPSILON ||
              bdgBox.max[i] < correctBdgBox->max[i] - EPSILON) {

            flag = false;

          }

        }

        // If the bounding box is the expected one
        if (flag == true) {

          // Display information
          printf("Succeed\n");

        // Else, the bounding box wasn't the expected one
        } else {

          // Display information
          printf("Failed\n");
          printf("Expected : ");
          AABB3DPrint(correctBdgBox);
          printf("\n");
          printf("     Got : ");
          AABB3DPrint(&bdgBox);
          printf("\n");

          // Terminate the unit tests
          exit(0);

        }

      // Else the Frames were not intersected,
      // no need to check the bounding box
      } else {

        // Display information
        printf(" Succeed\n");

      }

    }
    printf("\n");

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

  }

}

void Test3D(void) {

  // Declare two variables to memozie the arguments to the
  // Validation function
  Param3D paramP;
```

```
Param3D paramQ;

// Declare a variable to memorize the correct bounding box
AABB3D correctBdgBox;

// Execute the unit test on various cases

// ---------------------------
paramP = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
correctBdgBox = (AABB3D)
  {.min = {0.0, 0.0, 0.0},
   .max = {1.0, 1.0, 1.0}
  };
UnitTest3D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.5, 0.5, 0.5},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
correctBdgBox = (AABB3D)
  {.min = {0.5, 0.5, 0.5},
   .max = {1.0, 1.0, 1.0}
  };
UnitTest3D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
```

```
paramP = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
  {.type = FrameCuboid,
   .orig = {-0.5, -0.5, -0.5},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
correctBdgBox = (AABB3D)
  {.min = {0.0, 0.0, 0.0},
   .max = {0.5, 0.5, 0.5}
  };
UnitTest3D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
  {.type = FrameCuboid,
   .orig = {1.5, 1.5, 1.5},
   .comp =
     {{-1.0, 0.0, 0.0},
      {0.0, -1.0, 0.0},
      {0.0, 0.0, -1.0}}
  };
correctBdgBox = (AABB3D)
  {.min = {0.5, 0.5, 0.5},
   .max = {1.0, 1.0, 1.0}
  };
UnitTest3D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
```

```
paramQ = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.5, 1.5, -1.5},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, -1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
UnitTest3D(
  paramP,
  paramQ,
  false,
  NULL);

// ----------------------------
paramP = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, -1.0}}
  };
paramQ = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.5, 1.5, -1.5},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, -1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
correctBdgBox = (AABB3D)
  {.min = {0.5, 0.5, -1.0},
   .max = {1.0, 1.0, -0.5}
  };
UnitTest3D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param3D)
  {.type = FrameCuboid,
   .orig = {-1.01, -1.01, -1.01},
   .comp =
     {{1.0, 0.0, 0.0},
      {1.0, 1.0, 1.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
UnitTest3D(
  paramP,
  paramQ,
  false,
```

```
  NULL);

// ---------------------------
paramP = (Param3D)
  {.type = FrameCuboid,
   .orig = {-1.0, -1.0, -1.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {1.0, 1.0, 1.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, -0.5, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
correctBdgBox = (AABB3D)
  {.min = {0.0, -0.5, 0.0},
   .max = {1.0, 0.0, 1.0}
  };
UnitTest3D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param3D)
  {.type = FrameTetrahedron,
   .orig = {-1.0, -1.0, -1.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {1.0, 1.0, 1.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
  {.type = FrameCuboid,
   .orig = {0.0, -0.5, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
UnitTest3D(
  paramP,
  paramQ,
  false,
  NULL);

// ---------------------------
paramP = (Param3D)
  {.type = FrameCuboid,
   .orig = {-1.0, -1.0, -1.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {1.0, 1.0, 1.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
```

```
  {.type = FrameTetrahedron,
   .orig = {0.0, -0.5, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
correctBdgBox = (AABB3D)
  {.min = {0.0, -0.5, 0.0},
   .max = {0.75, 0.0, 0.75}
  };
UnitTest3D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param3D)
  {.type = FrameTetrahedron,
   .orig = {-1.0, -1.0, -1.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {1.0, 1.0, 1.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
  {.type = FrameTetrahedron,
   .orig = {0.0, -0.5, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
UnitTest3D(
  paramP,
  paramQ,
  false,
  NULL);

// ---------------------------
paramP = (Param3D)
  {.type = FrameTetrahedron,
   .orig = {-0.5, -1.0, -0.5},
   .comp =
     {{1.0, 0.0, 0.0},
      {1.0, 1.0, 1.0},
      {0.0, 0.0, 1.0}}
  };
paramQ = (Param3D)
  {.type = FrameTetrahedron,
   .orig = {0.0, -0.5, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}}
  };
correctBdgBox = (AABB3D)
  {.min = {0.0, -0.5, 0.0},
   .max = {0.5, -0.5 + 1.0 / 3.0, 0.5}
  };
UnitTest3D(
```

```
    paramP,
    paramQ,
    true,
    &correctBdgBox);


  // If we reached here, it means all the unit tests succeed
  printf("All unit tests 3D have succeed.\n");

}

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

  Test3D();

  return 0;
}
```

6.1.3 2D dynamic

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

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

// Epslon for numerical precision
#define EPSILON 0.0001

// Helper structure to pass arguments to the UnitTest function
typedef struct {
  FrameType type;
  double orig[2];
  double comp[2][2];
  double speed[2];
} Param2DTime;

// Unit test function
// Takes two Frame definitions, the correct answer in term of
// intersection/no intersection and the correct bounding box
// Run the FMB intersection detection alogirhtm on the Frames
// and check against the correct results
void UnitTest2DTime(
        const Param2DTime paramP,
        const Param2DTime paramQ,
         const bool correctAnswer,
  const AABB2DTime* const correctBdgBox) {

  // 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);

// Declare a variable to memorize the resulting bounding box
AABB2DTime bdgBoxLocal;

// 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--;) {

  // Display the tested frames
  Frame2DTimePrint(that);
  printf("\nagainst\n");
  Frame2DTimePrint(tho);
  printf("\n");

  // Run the FMB intersection test
  bool isIntersecting =
    FMBTestIntersection2DTime(
      that,
      tho,
      &bdgBoxLocal);

  // If the test hasn't given the expected answer about intersection
  if (isIntersecting != correctAnswer) {

    // Display information about the failure
    printf(" Failed\n");
    printf("Expected : ");
    if (correctAnswer == false)
      printf("no ");
    printf("intersection\n");
    printf("Got : ");
    if (isIntersecting == false)
      printf("no ");
    printf("intersection\n");
    exit(0);

  // Else, the test has given the expected answer about intersection
  } else {

    // If the Frames were intersecting
    if (isIntersecting == true) {

      AABB2DTime bdgBox;
      Frame2DTimeExportBdgBox(
        tho,
        &bdgBoxLocal,
        &bdgBox);
      // Check the bounding box
      bool flag = true;
      for (int i = 3;
           i--;) {

        if (bdgBox.min[i] > correctBdgBox->min[i] + EPSILON ||
            bdgBox.max[i] < correctBdgBox->max[i] - EPSILON) {
```

```
            flag = false;

          }

        }

        // If the bounding box is the expected one
        if (flag == true) {

          // Display information
          printf("Succeed\n");

        // Else, the bounding box wasn't the expected one
        } else {

          // Display information
          printf("Failed\n");
          printf("Expected : ");
          AABB2DTimePrint(correctBdgBox);
          printf("\n");
          printf("     Got : ");
          AABB2DTimePrint(&bdgBox);
          printf("\n");

          // Terminate the unit tests
          exit(0);

        }

      // Else the Frames were not intersected,
      // no need to check the bounding box
      } else {

        // Display information
        printf(" Succeed\n");

      }

    }
    printf("\n");

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

  }

}

void Test2DTime(void) {

  // Declare two variables to memozie the arguments to the
  // Validation function
  Param2DTime paramP;
  Param2DTime paramQ;

  // Declare a variable to memorize the correct bounding box
  AABB2DTime correctBdgBox;

  // Execute the unit test on various cases
```

```
// ---------------------------
paramP = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}},
   .speed = {0.0, 0.0}
  };
paramQ = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {-1.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}},
   .speed = {-1.0, 0.0}
  };
UnitTest2DTime(
  paramP,
  paramQ,
  false,
  NULL);

// ---------------------------
paramP = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}},
   .speed = {0.0, 0.0}
  };
paramQ = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {-1.01, -1.01},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}},
   .speed = {1.0, 0.0}
  };
UnitTest2DTime(
  paramP,
  paramQ,
  false,
  NULL);

// ---------------------------
paramP = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}},
   .speed = {0.0, 0.0}
  };
paramQ = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {-1.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}},
   .speed = {1.0, 0.0}
```

```
  };
correctBdgBox = (AABB2DTime)
  {.min = {0.0, 0.0, 0.0},
   .max = {1.0, 1.0, 1.0}
  };
UnitTest2DTime(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}},
   .speed = {0.0, 0.0}
  };
paramQ = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {-1.0, 0.25},
   .comp =
     {{0.5, 0.0},
      {0.0, 0.5}},
   .speed = {4.0, 0.0}
  };
correctBdgBox = (AABB2DTime)
  {.min = {0.0, 0.25, 0.125},
   .max = {1.0, 0.75, 0.5}
  };
UnitTest2DTime(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}},
   .speed = {0.0, 0.0}
  };
paramQ = (Param2DTime)
  {.type = FrameCuboid,
   .orig = {0.25, -1.0},
   .comp =
     {{0.5, 0.0},
      {0.0, 0.5}},
   .speed = {0.0, 4.0}
  };
correctBdgBox = (AABB2DTime)
  {.min = {0.25, 0.0, 0.125},
   .max = {0.75, 1.0, 0.5}
  };
UnitTest2DTime(
  paramP,
  paramQ,
```

```
    true,
    &correctBdgBox);

  // ---------------------------
  paramP = (Param2DTime)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0},
     .comp =
       {{1.0, 0.0},
        {0.0, 1.0}},
     .speed = {0.0, 0.0}
    };
  paramQ = (Param2DTime)
    {.type = FrameCuboid,
     .orig = {0.9, -1.0},
     .comp =
       {{0.5, 0.0},
        {0.0, 0.5}},
     .speed = {0.0, 4.0}
    };
  correctBdgBox = (AABB2DTime)
    {.min = {0.9, 0.0, 0.125},
     .max = {1.0, 1.0, 0.5}
    };
  UnitTest2DTime(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

  // If we reached here, it means all the unit tests succeed
  printf("All unit tests 2DTime have succeed.\n");

}

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

  Test2DTime();

  return 0;
}
```

6.1.4 3D dynamic

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

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

// Epslon for numerical precision
#define EPSILON 0.0001

// Helper structure to pass arguments to the UnitTest function
typedef struct {
  FrameType type;
  double orig[3];
  double comp[3][3];
```

```
  double speed[3];
} Param3DTime;

// Unit test function
// Takes two Frame definitions, the correct answer in term of
// intersection/no intersection and the correct bounding box
// Run the FMB intersection detection alogirhtm on the Frames
// and check against the correct results
void UnitTest3DTime(
        const Param3DTime paramP,
        const Param3DTime paramQ,
         const bool correctAnswer,
  const AABB3DTime* const correctBdgBox) {

  // 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);

  // Declare a variable to memorize the resulting bounding box
  AABB3DTime bdgBoxLocal;

  // 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--;) {

    // Display the tested frames
    Frame3DTimePrint(that);
    printf("\nagainst\n");
    Frame3DTimePrint(tho);
    printf("\n");

    // Run the FMB intersection test
    bool isIntersecting =
      FMBTestIntersection3DTime(
        that,
        tho,
        &bdgBoxLocal);

    // If the test hasn't given the expected answer about intersection
    if (isIntersecting != correctAnswer) {

      // Display information about the failure
      printf(" Failed\n");
      printf("Expected : ");
      if (correctAnswer == false)
        printf("no ");
      printf("intersection\n");
```

```
  printf("Got : ");
  if (isIntersecting == false)
    printf("no ");
  printf("intersection\n");
  exit(0);

// Else, the test has given the expected answer about intersection
} else {

  // If the Frames were intersecting
  if (isIntersecting == true) {

    AABB3DTime bdgBox;
    Frame3DTimeExportBdgBox(
      tho,
      &bdgBoxLocal,
      &bdgBox);
    // Check the bounding box
    bool flag = true;
    for (int i = 4;
         i--;) {

      if (bdgBox.min[i] > correctBdgBox->min[i] + EPSILON ||
          bdgBox.max[i] < correctBdgBox->max[i] - EPSILON) {

        flag = false;

      }

    }

    // If the bounding box is the expected one
    if (flag == true) {

      // Display information
      printf("Succeed\n");

    // Else, the bounding box wasn't the expected one
    } else {

      // Display information
      printf("Failed\n");
      printf("Expected : ");
      AABB3DTimePrint(correctBdgBox);
      printf("\n");
      printf("     Got : ");
      AABB3DTimePrint(&bdgBox);
      printf("\n");

      // Terminate the unit tests
      exit(0);

    }

  // Else the Frames were not intersected,
  // no need to check the bounding box
  } else {

    // Display information
    printf(" Succeed\n");

  }
```

```
    }
    printf("\n");

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

  }

}

void Test3DTime(void) {

  // Declare two variables to memozie the arguments to the
  // Validation function
  Param3DTime paramP;
  Param3DTime paramQ;

  // Declare a variable to memorize the correct bounding box
  AABB3DTime correctBdgBox;

  // Execute the unit test on various cases

  // ---------------------------
  paramP = (Param3DTime)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0, 0.0},
     .comp =
       {{1.0, 0.0, 0.0},
        {0.0, 1.0, 0.0},
        {0.0, 0.0, 1.0}},
     .speed = {0.0, 0.0, 0.0}
    };
  paramQ = (Param3DTime)
    {.type = FrameCuboid,
     .orig = {-1.0, 0.0, 0.0},
     .comp =
       {{1.0, 0.0, 0.0},
        {0.0, 1.0, 0.0},
        {0.0, 0.0, 1.0}},
     .speed = {-1.0, 0.0, 0.0}
    };
  UnitTest3DTime(
    paramP,
    paramQ,
    false,
    NULL);

  // ---------------------------
  paramP = (Param3DTime)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0, 0.0},
     .comp =
       {{1.0, 0.0, 0.0},
        {0.0, 1.0, 0.0},
        {0.0, 0.0, 1.0}},
     .speed = {0.0, 0.0, 0.0}
    };
  paramQ = (Param3DTime)
    {.type = FrameCuboid,
     .orig = {-1.01, -1.01, 0.0},
```

```
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}},
   .speed = {1.0, 0.0, 0.0}
  };
UnitTest3DTime(
  paramP,
  paramQ,
  false,
  NULL);

// ---------------------------
paramP = (Param3DTime)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}},
   .speed = {0.0, 0.0, 0.0}
  };
paramQ = (Param3DTime)
  {.type = FrameCuboid,
   .orig = {-1.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}},
   .speed = {1.0, 0.0, 0.0}
  };
correctBdgBox = (AABB3DTime)
  {.min = {0.0, 0.0, 0.0, 0.0},
   .max = {1.0, 1.0, 1.0, 1.0}
  };
UnitTest3DTime(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param3DTime)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}},
   .speed = {0.0, 0.0, 0.0}
  };
paramQ = (Param3DTime)
  {.type = FrameCuboid,
   .orig = {-1.0, 0.25, 0.0},
   .comp =
     {{0.5, 0.0, 0.0},
      {0.0, 0.5, 0.0},
      {0.0, 0.0, 1.0}},
   .speed = {4.0, 0.0, 0.0}
  };
correctBdgBox = (AABB3DTime)
  {.min = {0.0, 0.25, 0.0, 0.125},
```

```
    .max = {1.0, 0.75, 1.0, 0.5}
  };
UnitTest3DTime(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param3DTime)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}},
   .speed = {0.0, 0.0, 0.0}
  };
paramQ = (Param3DTime)
  {.type = FrameCuboid,
   .orig = {0.25, -1.0, 0.0},
   .comp =
     {{0.5, 0.0, 0.0},
      {0.0, 0.5, 0.0},
      {0.0, 0.0, 1.0}},
   .speed = {0.0, 4.0, 0.0}
  };
correctBdgBox = (AABB3DTime)
  {.min = {0.25, 0.0, 0.0, 0.125},
   .max = {0.75, 1.0, 1.0, 0.5}
  };
UnitTest3DTime(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ---------------------------
paramP = (Param3DTime)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0, 0.0},
   .comp =
     {{1.0, 0.0, 0.0},
      {0.0, 1.0, 0.0},
      {0.0, 0.0, 1.0}},
   .speed = {0.0, 0.0, 0.0}
  };
paramQ = (Param3DTime)
  {.type = FrameCuboid,
   .orig = {0.9, -1.0, 0.0},
   .comp =
     {{0.5, 0.0, 0.0},
      {0.0, 0.5, 0.0},
      {0.0, 0.0, 1.0}},
   .speed = {0.0, 4.0, 0.0}
  };
correctBdgBox = (AABB3DTime)
  {.min = {0.9, 0.0, 0.0, 0.125},
   .max = {1.0, 1.0, 1.0, 0.5}
  };
UnitTest3DTime(
  paramP,
```

```
    paramQ,
    true,
    &correctBdgBox);

  // If we reached here, it means all the unit tests succeed
  printf("All unit tests 3DTime have succeed.\n");

}

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

  Test3DTime();

  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 ValidationOnePair2D(
  const Param2D paramP,
  const Param2D paramQ) {

  // Create the two Frames
  Frame2D P =
    Frame2DCreateStatic(
      paramP.type,
```

```
    paramP.orig,
    paramP.comp);

Frame2D Q =
  Frame2DCreateStatic(
    paramQ.type,
    paramQ.orig,
    paramQ.comp);

// Helper variables to loop on the pair (that, tho) and (tho, that)
Frame2D* that = &P;
Frame2D* tho = &Q;

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

  // Test intersection with FMB
  bool isIntersectingFMB =
    FMBTestIntersection2D(
      that,
      tho,
      NULL);

  // Test intersection with SAT
  bool isIntersectingSAT =
    SATTestIntersection2D(
      that,
      tho);

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

    // Print the disagreement
    printf("Validation2D has failed\n");
    Frame2DPrint(that);
    printf(" against ");
    Frame2DPrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT == false)
      printf("no ");
    printf("intersection\n");

    // Stop the validation
    exit(0);

  }

  // If the Frames are in intersection
  if (isIntersectingFMB == true) {

    // Update the number of intersection
    nbInter++;

  // If the Frames are not in intersection
  } else {
```

```
      // Update the number of no intersection
      nbNoInter++;

    }

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

  }

}

void 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
      ValidationOnePair2D(
        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 ValidationOnePair3D(
  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
      ValidationOnePair3D(
        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 the 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 ValidationOnePair2DTime(
  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;

}
```

```
}

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
      ValidationOnePair2DTime(
        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 the 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 ValidationOnePair3DTime(
  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;

  }

}

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;
```

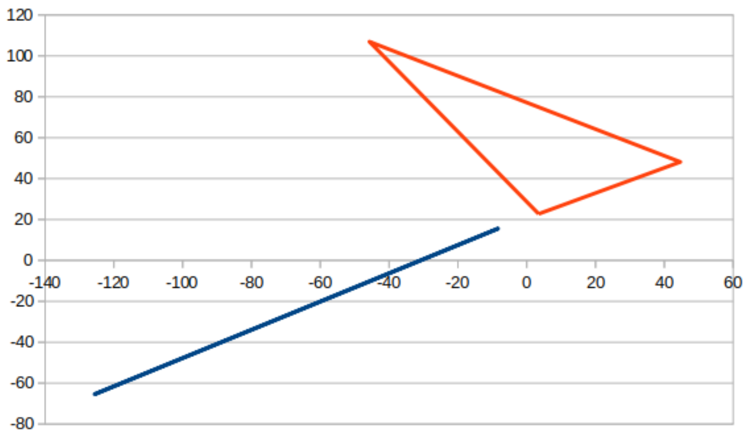
```
            iComp--;) {

          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
      ValidationOnePair3DTime(
        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 are 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 469098 intersections and 1530820 no intersections
```

7.2.3 2D dynamic

```
===== 2D dynamic ======
Validation2DTime has succeed.
Tested 744820 intersections and 1255100 no intersections
```

7.2.4 3D static

```
===== 3D static ======
Validation3D has succeed.
Tested 315664 intersections and 1684334 no intersections
```

7.2.5 3D dynamic

```
===== 3D dynamic ======
Validation3DTime has succeed.
Tested 523890 intersections and 1476110 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 memorize 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 memorize 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 memorize 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 memorize 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

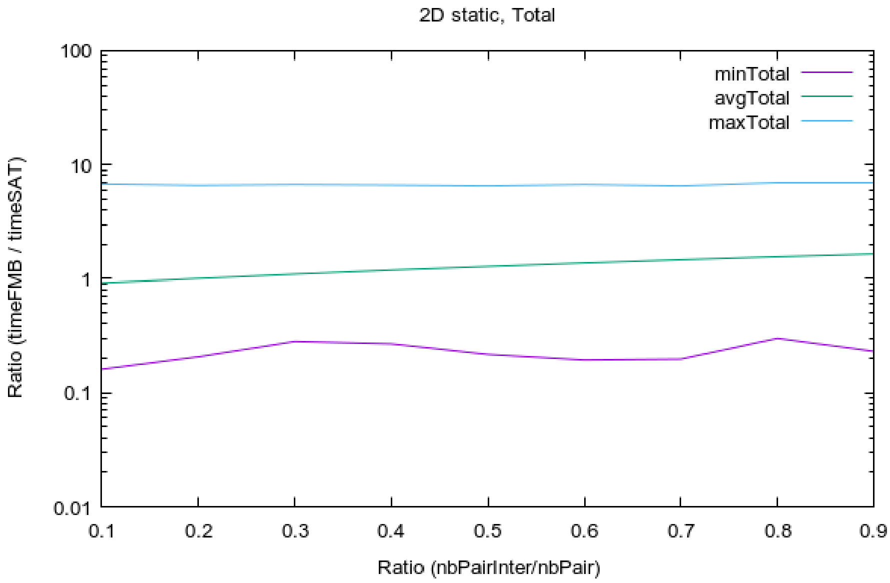
In this subsection I give the results of the qualification for each case. These results are commented in the next section.

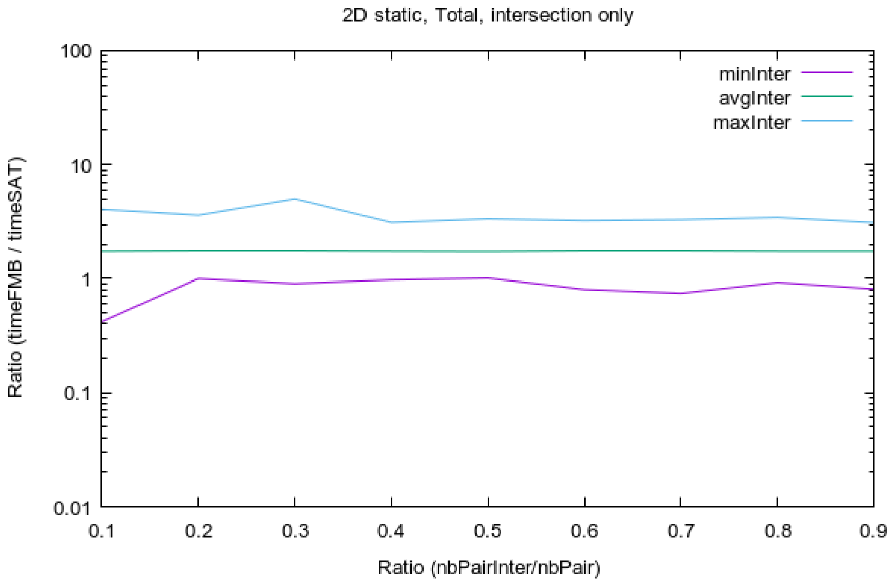
8.2.1 2D static

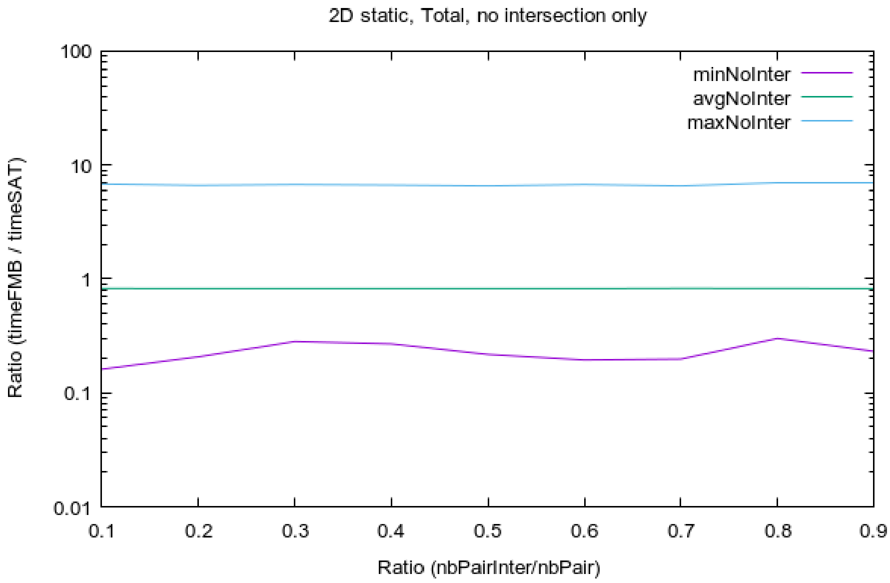
```
percPairInter   countInter      countNoInter    minInter        avgInter
            maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    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
```

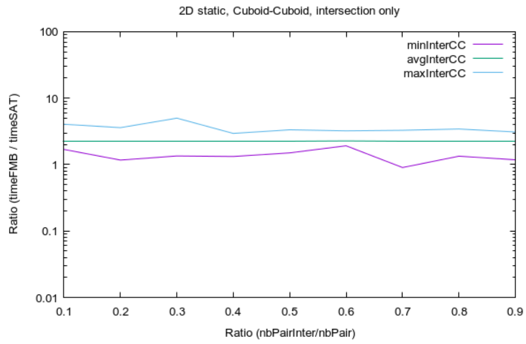
```
    maxInterTT      minNoInterTT    avgNoInterTT    maxNoInterTT
    minTotalTT      avgTotalTT      maxTotalTT
0.1     46878   153112  0.418972        1.740664        4.038462
    0.160377        0.823746        6.785714        0.160377        0.915438
            6.785714        13290   36906   1.690141        2.266804
    4.038462        0.225000        0.777149        3.275000        0.225000
            0.926115        4.038462        11594   38754   0.418972
    1.676619        3.689655        0.160377        0.827159        6.785714
            0.160377        0.912105        6.785714        11722   38250
    0.886792        1.670610        2.396552        0.240741        0.828076
            6.666667        0.240741        0.912329        6.666667
    10272   39202   0.614379        1.212170        1.925373        0.164179
            0.860015        6.428571        0.164179        0.895230
    6.428571
0.2     47322   152666  1.000000        1.736314        3.596154
    0.206349        0.821453        6.600000        0.206349        1.004425
            6.600000        13214   37188   1.168317        2.268245
    3.596154        0.325000        0.776310        3.282051        0.325000
            1.074697        3.596154        11630   38020   1.301205
    1.676722        2.175439        0.259259        0.828660        6.466667
            0.259259        0.998273        6.466667        11814   37914
    1.000000        1.671872        2.298246        0.206349        0.821100
            6.600000        0.206349        0.991255        6.600000
    10664   39544   1.036585        1.213569        1.514286        0.279070
            0.857316        6.285714        0.279070        0.928566
    6.285714
0.3     47226   152766  0.902174        1.737370        5.000000
    0.282051        0.822148        6.714286        0.282051        1.096715
            6.714286        13160   37412   1.348315        2.265935
    5.000000        0.282051        0.778870        3.875000        0.282051
            1.224989        5.000000        11862   38048   1.009434
    1.676124        2.698413        0.309524        0.826288        6.533333
            0.309524        1.081239        6.533333        11834   37696
    1.000000        1.670316        3.696203        0.295455        0.827593
            6.714286        0.295455        1.080410        6.714286
    10370   39610   0.902174        1.213176        2.500000        0.292683
            0.853866        6.142857        0.292683        0.961659
    6.142857
0.4     46854   153138  0.976744        1.741146        3.105263
    0.268293        0.821337        6.642857        0.268293        1.189260
            6.642857        13110   37276   1.322581        2.267323
    2.943396        0.378378        0.777075        3.725000        0.378378
            1.373174        3.725000        11826   38234   1.028846
    1.676494        2.803279        0.275000        0.828659        6.642857
            0.275000        1.167793        6.642857        11874   38014
    1.043478        1.671352        3.105263        0.268293        0.825977
            6.533333        0.268293        1.164127        6.533333
    10044   39614   0.976744        1.212982        1.514286        0.295455
            0.851467        6.214286        0.295455        0.996073
    6.214286
0.5     47014   152974  1.011905        1.731075        3.339623
    0.216667        0.822552        6.533333        0.216667        1.276813
            6.533333        12826   36442   1.500000        2.265458
    3.339623        0.355263        0.778023        3.589744        0.355263
            1.521741        3.589744        11846   38394   1.084211
    1.676002        2.409836        0.216667        0.827124        6.466667
            0.216667        1.251563        6.466667        11750   38416
    1.144578        1.670519        2.253968        0.333333        0.826234
            6.533333        0.333333        1.248376        6.533333
    10592   39722   1.011905        1.212751        1.573529        0.250000
            0.855424        6.214286        0.250000        1.034088
    6.214286
```

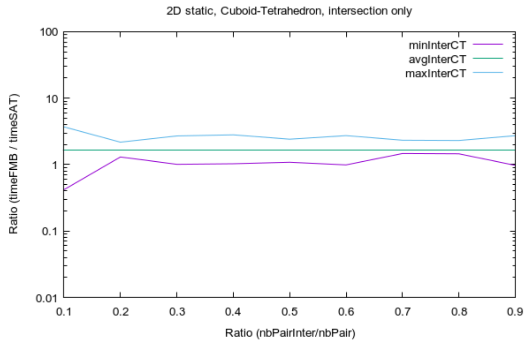
```
0.6     47172   152828  0.801887        1.739365        3.222222
    0.193548        0.821359        6.714286        0.193548        1.372163
            6.714286        13196   36788   1.919355        2.268979
    3.222222        0.230769        0.773747        3.333333        0.230769
            1.670886        3.333333        11798   38362   0.989691
    1.676794        2.728814        0.315789        0.828230        6.714286
            0.315789        1.337368        6.714286        11782   37978
    1.402985        1.672582        2.887097        0.193548        0.827085
            6.600000        0.193548        1.334383        6.600000
    10396   39700   0.801887        1.213805        1.704225        0.300000
            0.853362        6.285714        0.300000        1.069628
    6.285714
0.7     47084   152908  0.743363        1.739790        3.283019
    0.196970        0.824594        6.533333        0.196970        1.465231
            6.533333        13230   36584   0.902985        2.268273
    3.283019        0.200000        0.776531        3.210526        0.200000
            1.820750        3.283019        11588   38282   1.468750
    1.676866        2.327586        0.280000        0.830210        6.533333
            0.280000        1.422869        6.533333        11898   38420
    0.940000        1.672581        2.620690        0.196970        0.832646
            6.533333        0.196970        1.420600        6.533333
    10368   39622   0.743363        1.212879        2.000000        0.275000
            0.855739        6.285714        0.275000        1.105737
    6.285714
0.8     47272   152722  0.923913        1.741554        3.431373
    0.300000        0.823731        6.928571        0.300000        1.557989
            6.928571        13416   36454   1.337079        2.266165
    3.431373        0.301887        0.774009        3.263158        0.301887
            1.967734        3.431373        12042   37920   1.453125
    1.676074        2.306452        0.300000        0.837903        6.928571
            0.300000        1.508440        6.928571        11562   38472
    1.223404        1.669931        2.140351        0.324324        0.827423
            6.666667        0.324324        1.501430        6.666667
    10252   39876   0.923913        1.212722        1.594203        0.307692
            0.852147        6.214286        0.307692        1.140607
    6.214286
0.9     46772   153222  0.811321        1.739447        3.096154
    0.230769        0.820697        6.928571        0.230769        1.647572
            6.928571        13040   36934   1.179245        2.267521
    3.096154        0.239130        0.778663        3.210526        0.239130
            2.118635        3.210526        11846   38464   0.972973
    1.676054        2.721311        0.292683        0.823071        6.466667
            0.292683        1.590756        6.466667        11764   37940
    1.363636        1.671707        2.338710        0.255319        0.827524
            6.928571        0.255319        1.587289        6.928571
    10122   39884   0.811321        1.212059        1.536232        0.230769
            0.850839        6.214286        0.230769        1.175937
    6.214286
```

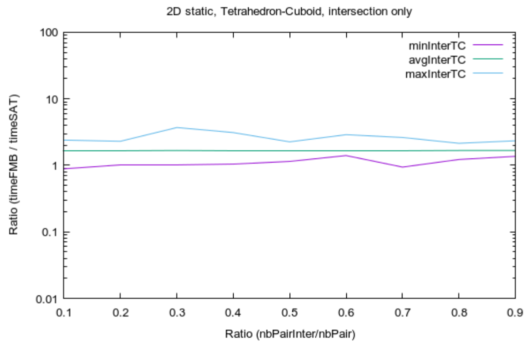

2D static, Total
minTotal
avgTotal
maxTotal
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

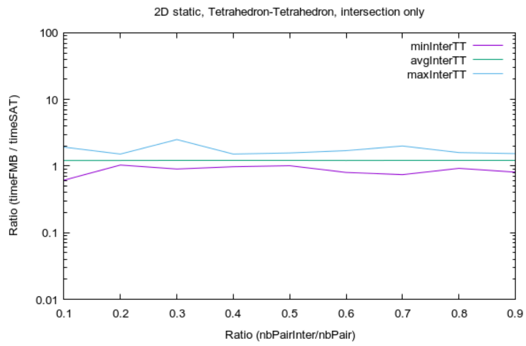

2D static, Total, intersection only
minInter
avgInter
maxInter
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

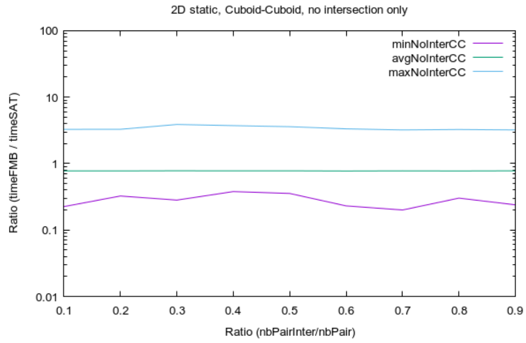

2D static, Total, no intersection only
minNoInter
avgNoInter
maxNoInter
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

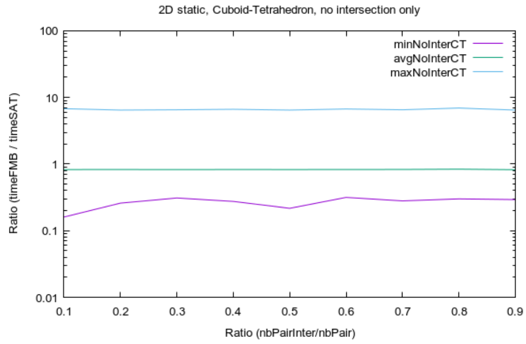

2D static, Cuboid-Cuboid, intersection only
minInterCC
avgInterCC
maxInterCC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

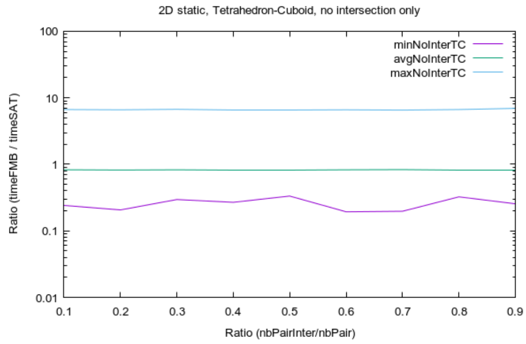

2D static, Cuboid-Tetrahedron, intersection only
minInterCT
avgInterCT
maxInterCT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

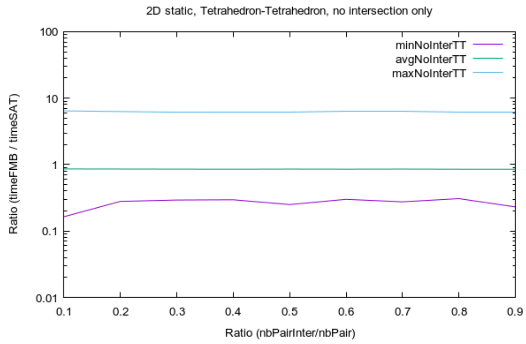

2D static, Tetrahedron-Cuboid, intersection only
minInterTC
avgInterTC
maxInterTC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


2D static, Tetrahedron-Tetrahedron, intersection only
minInterTT
avgInterTT
maxInterTT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


2D static, Cuboid-Cuboid, no intersection only
minNoInterCC
avgNoInterCC
maxNoInterCC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


2D static, Cuboid-Tetrahedron, no intersection only
minNoInterCT
avgNoInterCT
maxNoInterCT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


2D static, Tetrahedron-Cuboid, no intersection only
minNoInterTC
avgNoInterTC
maxNoInterTC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

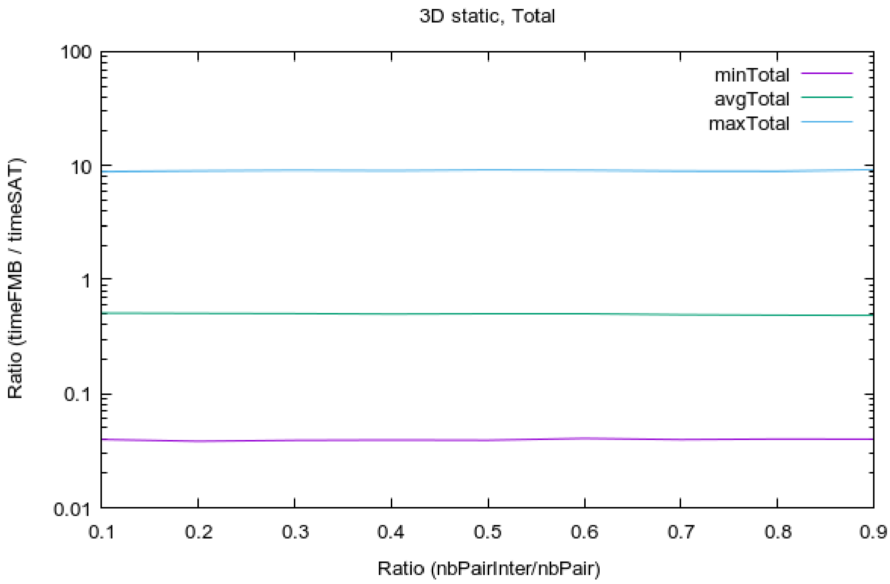

2D static, Tetrahedron-Tetrahedron, no intersection only
minNoInterTT
avgNoInterTT
maxNoInterTT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

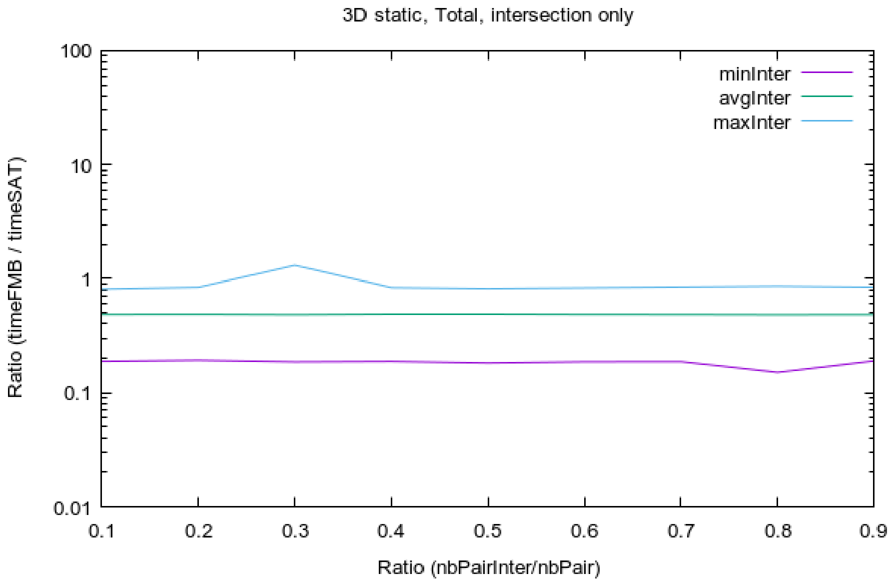
8.2.2 3D static

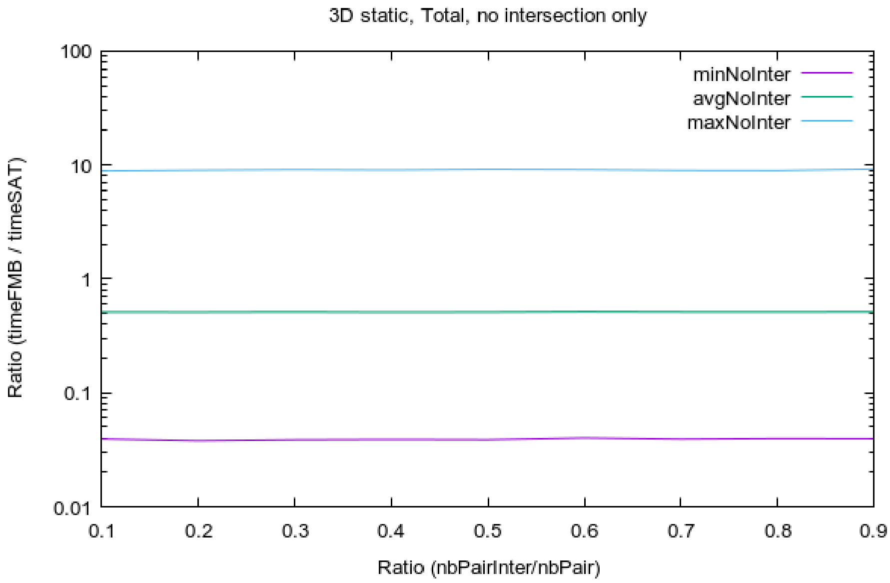
```
percPairInter   countInter      countNoInter    minInter        avgInter
        maxInter        minNoInter      avgNoInter      maxNoInter
   minTotal        avgTotal        maxTotal        countInterCC
   countNoInterCC  minInterCC      avgInterCC      maxInterCC
   minNoInterCC    avgNoInterCC    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     31460   168538  0.187793        0.483440        0.809174
   0.039416        0.511192        8.870968        0.039416        0.508417
         8.870968        10576   39376   0.600000        0.714782
   0.809174        0.063736        0.364478        3.000000        0.063736
         0.399509        3.000000        7914    42194   0.246626
   0.410338        0.632743        0.044800        0.495692        8.870968
         0.044800        0.487157        8.870968        7810    42712
   0.266216        0.410165        0.549275        0.045528        0.495379
         8.593750        0.045528        0.486857        8.593750
   5160    44256   0.187793        0.232305        0.314951        0.039416
         0.671769        8.208333        0.039416        0.627823
   8.208333
0.2     31374   168626  0.191895        0.484779        0.839572
   0.038028        0.510353        9.000000        0.038028        0.505238
         9.000000        10682   39532   0.630178        0.714480
   0.839572        0.062084        0.362791        2.877551        0.062084
         0.433129        2.877551        7764    42204   0.226358
   0.410275        0.557692        0.045016        0.494326        9.000000
         0.045016        0.477516        9.000000        7802    41784
   0.262599        0.410466        0.582596        0.044234        0.492460
         8.838710        0.044234        0.476061        8.838710
   5126    45106   0.191895        0.232060        0.289082        0.038028
         0.671252        8.291667        0.038028        0.583413
   8.291667
0.3     31588   168412  0.186047        0.481205        1.311151
   0.038793        0.512643        9.096774        0.038793        0.503212
         9.096774        10442   39728   0.447219        0.714673
   1.311151        0.063205        0.364349        2.761905        0.063205
         0.469446        2.761905        8042    42208   0.269841
   0.410498        0.592697        0.046129        0.502061        9.096774
         0.046129        0.474592        9.096774        7812    41806
   0.246041        0.410765        0.737084        0.045016        0.498671
         9.000000        0.045016        0.472299        9.000000
   5292    44670   0.186047        0.231964        0.317419        0.038793
         0.667606        8.208333        0.038793        0.536914
   8.208333
0.4     31294   168706  0.187500        0.482227        0.834545
   0.039017        0.509375        9.032258        0.039017        0.498516
         9.032258        10452   39872   0.619122        0.714860
   0.834545        0.063877        0.361042        2.739726        0.063877
         0.502569        2.739726        7664    41746   0.264865
   0.410403        0.597345        0.044444        0.497419        8.967742
         0.044444        0.462612        8.967742        7920    42688
   0.264744        0.410636        0.574594        0.045234        0.497144
         9.032258        0.045234        0.462541        9.032258
```

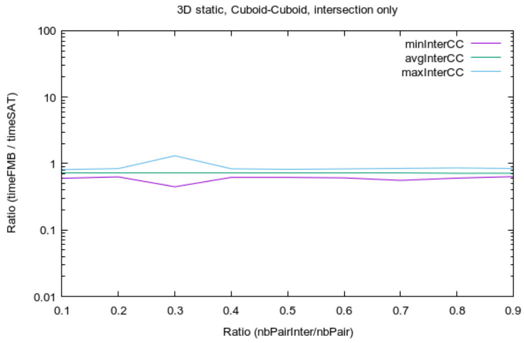
```
    5258    44400   0.187500        0.232321        0.273203        0.039017
            0.665581        8.208333        0.039017        0.492277
    8.208333
0.5     31762   168238  0.181723        0.482333        0.816697
    0.038849        0.510855        9.161290        0.038849        0.496594
            9.161290        10652   39232   0.618524        0.715150
    0.816697        0.062361        0.361938        2.738255        0.062361
            0.538544        2.738255        7894    41848   0.265943
    0.410370        0.616690        0.043887        0.497869        8.757576
            0.043887        0.454119        8.757576        7818    42454
    0.264430        0.410520        0.556716        0.045378        0.497332
            9.161290        0.045378        0.453926        9.161290
    5398    44704   0.181723        0.232155        0.303433        0.038849
            0.666544        8.125000        0.038849        0.449350
    8.125000
0.6     31572   168428  0.186240        0.483558        0.831560
    0.040299        0.514643        9.096774        0.040299        0.495992
            9.096774        10676   39336   0.608964        0.714966
    0.831560        0.062366        0.362200        2.753425        0.062366
            0.573859        2.753425        7858    42206   0.256892
    0.410529        0.646707        0.046205        0.501007        9.096774
            0.046205        0.446720        9.096774        7730    41928
    0.259212        0.410645        0.609398        0.045677        0.497317
            8.781250        0.045677        0.445314        8.781250
    5308    44958   0.186240        0.232421        0.300771        0.040299
            0.676983        8.291667        0.040299        0.410246
    8.291667
0.7     31482   168518  0.186839        0.482851        0.845541
    0.039301        0.512373        8.967742        0.039301        0.491708
            8.967742        10486   39378   0.558205        0.715232
    0.845541        0.063927        0.365010        2.821918        0.063927
            0.610165        2.821918        7796    42290   0.265943
    0.410564        0.583221        0.043614        0.497514        8.750000
            0.043614        0.436649        8.750000        8014    42152
    0.256051        0.411203        0.640693        0.043818        0.497534
            8.967742        0.043818        0.437102        8.967742
    5186    44698   0.186839        0.232369        0.298765        0.039301
            0.670249        8.652174        0.039301        0.363733
    8.652174
0.8     31498   168502  0.150344        0.480703        0.859130
    0.039764        0.512065        8.935484        0.039764        0.486975
            8.935484        10496   39290   0.603687        0.714650
    0.859130        0.063736        0.364853        2.823129        0.063736
            0.644690        2.823129        7862    42206   0.268657
    0.409905        0.588028        0.044374        0.498538        8.483871
            0.044374        0.427632        8.483871        7686    42104
    0.254499        0.410428        0.676471        0.044515        0.498741
            8.935484        0.044515        0.428090        8.935484
    5454    44902   0.150344        0.231573        0.299742        0.039764
            0.666085        8.125000        0.039764        0.318475
    8.125000
0.9     31694   168306  0.189286        0.482025        0.842593
    0.039700        0.513070        9.193548        0.039700        0.485130
            9.193548        10554   39102   0.636215        0.714860
    0.842593        0.063596        0.363990        2.732877        0.063596
            0.679773        2.732877        7904    42240   0.266756
    0.410191        0.624093        0.044094        0.498129        8.967742
            0.044094        0.418985        8.967742        7950    42218
    0.266576        0.410547        0.691285        0.045307        0.498747
            9.193548        0.045307        0.419367        9.193548
    5286    44746   0.189286        0.232063        0.271782        0.039700
            0.670964        8.250000        0.039700        0.275953
```

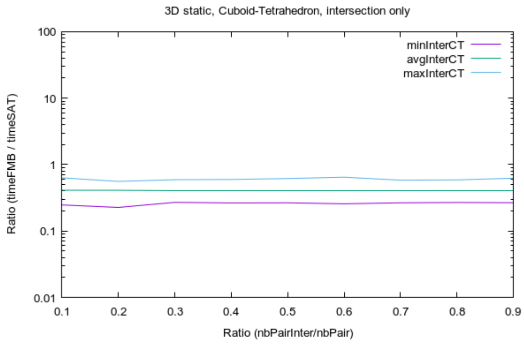
8.250000

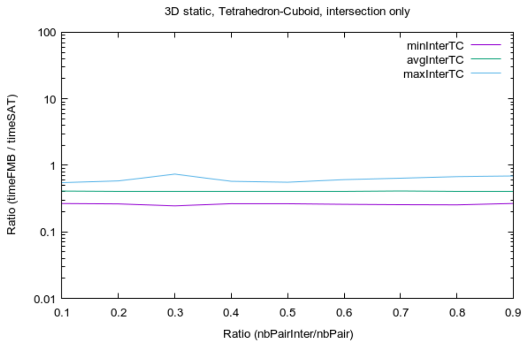

3D static, Total
minTotal
avgTotal
maxTotal
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

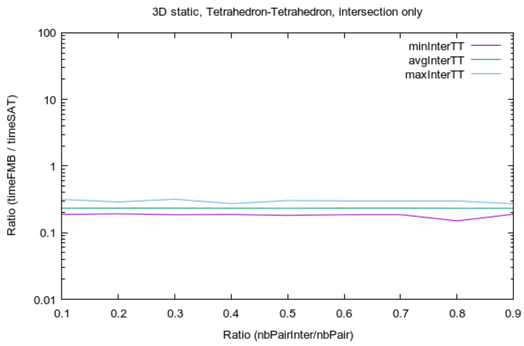

3D static, Total, intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minInter
avgInter
maxInter
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

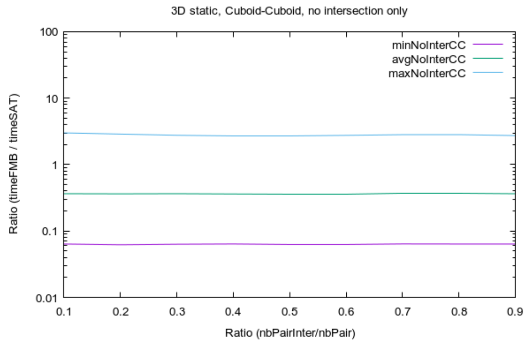

3D static, Total, no intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minNoInter
avgNoInter
maxNoInter
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

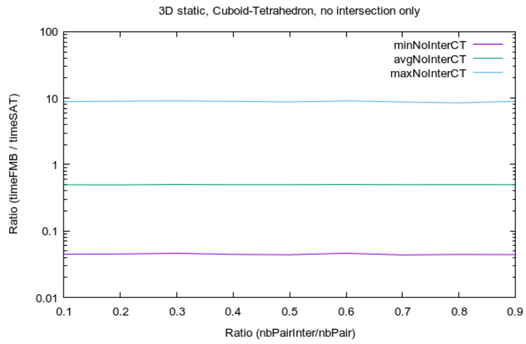

3D static, Cuboid-Cuboid, intersection only
minInterCC
avgInterCC
maxInterCC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

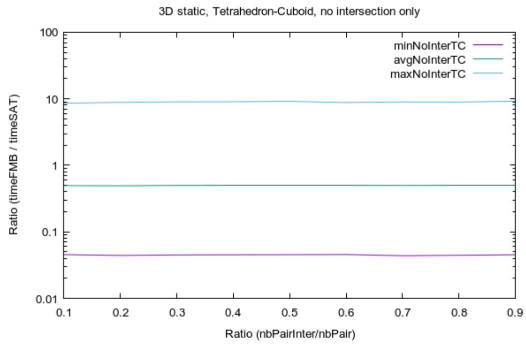

3D static, Cuboid-Tetrahedron, intersection only
minInterCT
avgInterCT
maxInterCT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

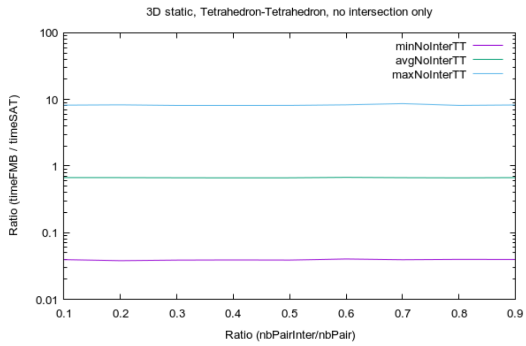

3D static, Tetrahedron-Cuboid, intersection only
minInterTC
avgInterTC
maxInterTC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


3D static, 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 static, Cuboid-Cuboid, no intersection only
minNoInterCC
avgNoInterCC
maxNoInterCC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


3D static, Cuboid-Tetrahedron, no intersection only
minNoInterCT
avgNoInterCT
maxNoInterCT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


3D static, Tetrahedron-Cuboid, no intersection only
minNoInterTC
avgNoInterTC
maxNoInterTC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

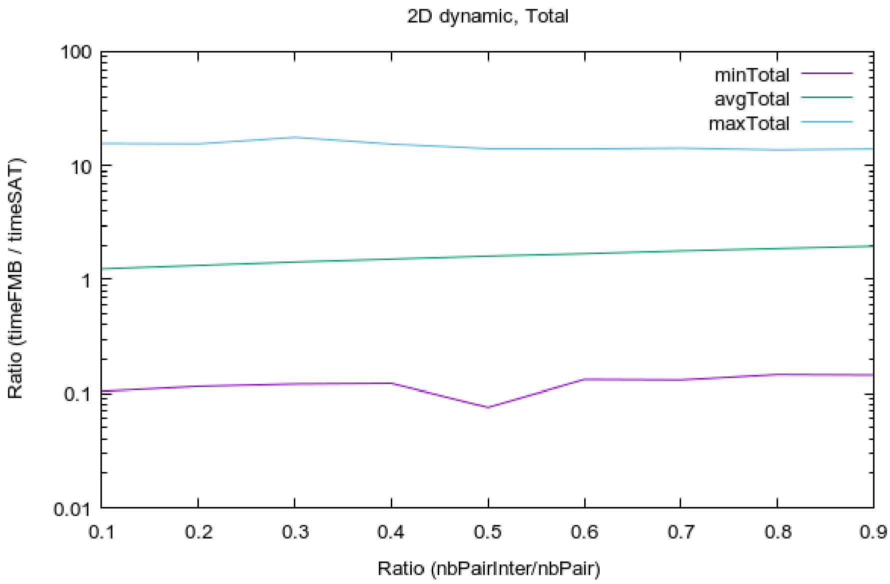

3D static, Tetrahedron-Tetrahedron, no intersection only
minNoInterTT
avgNoInterTT
maxNoInterTT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

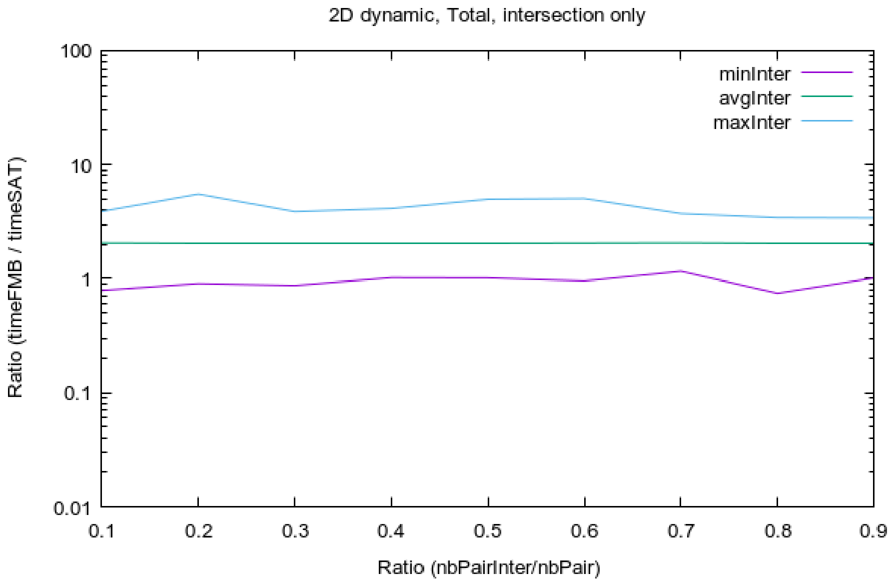
8.2.3 2D dynamic

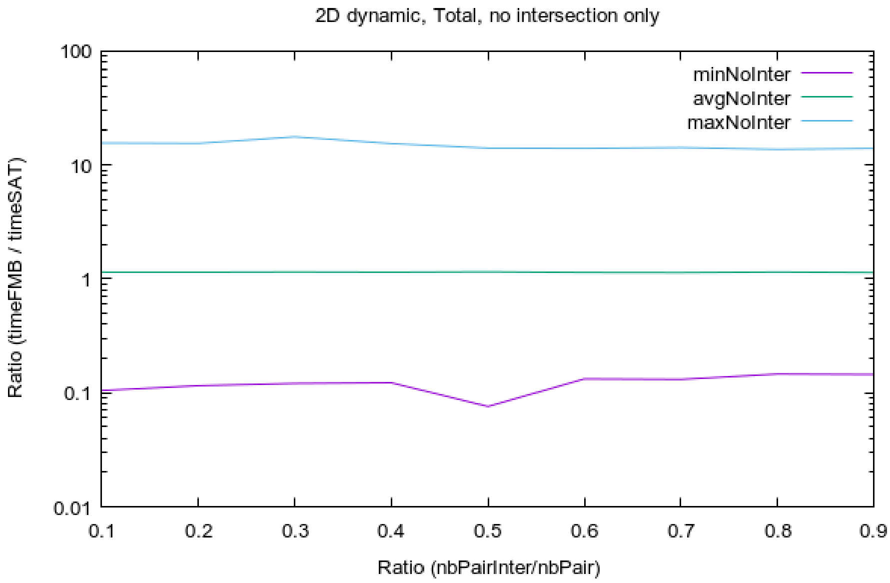
```
percPairInter   countInter      countNoInter    minInter        avgInter
            maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    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     74774   125220  0.789137        2.063885        3.890625
    0.104938        1.148536        15.666667       0.104938        1.240070
            15.666667       20228   29784   1.223776        2.674546
    3.890625        0.141791        1.132268        15.666667       0.141791
            1.286496        15.666667       18516   31804   1.142259
    1.998718        3.354610        0.104938        1.143461
    14.250000       0.104938        1.228986        14.250000       18794
    31066   1.123333        1.993644        2.851064        0.107595
    1.156510        14.000000       0.107595        1.240223
    14.000000       17236   32566   0.789137        1.493815        2.876543
            0.144330        1.160762        10.600000       0.144330
    1.194067        10.600000
0.2     74180   125816  0.903974        2.061758        5.515625
    0.115646        1.147415        15.535714       0.115646        1.330284
            15.535714       19882   30304   1.714286        2.674227
    5.515625        0.115646        1.138799        15.535714       0.115646
            1.445885        15.535714       18378   31342   1.120275
    1.998057        3.295775        0.140496        1.157490
    14.130435       0.140496        1.325603        14.130435       18764
    31412   0.903974        1.993557        2.775362        0.146789
    1.141950        14.375000       0.146789        1.312271
    14.375000       17156   32758   0.984000        1.494802        2.857143
            0.155963        1.150986        10.782609       0.155963
    1.219749        10.782609
0.3     74568   125412  0.869010        2.059995        3.874126
    0.121212        1.152711        17.758621       0.121212        1.424896
            17.758621       19730   30154   1.189831        2.674717
    3.874126        0.144144        1.166944        17.758621       0.144144
            1.619276        17.758621       19016   31210   0.869010
    1.997840        3.345070        0.142857        1.156546
    13.666667       0.142857        1.408934        13.666667       18650
    31102   1.382653        1.993774        2.862319        0.121212
    1.136842        13.625000       0.121212        1.393922
    13.625000       17172   32946   0.996154        1.494453        2.536145
            0.151515        1.151033        10.500000       0.151515
    1.254059        10.500000
0.4     74224   125766  1.022059        2.062423        4.129771
    0.122807        1.147212        15.481481       0.122807        1.513296
            15.481481       20000   30174   1.917127        2.674475
    4.129771        0.155963        1.138274        15.481481       0.155963
            1.752754        15.481481       18818   31212   1.022059
    1.997798        3.038674        0.128788        1.186267
    14.961538       0.128788        1.510880        14.961538       18206
    31744   1.092742        1.993728        2.774390        0.122807
    1.136914        13.880000       0.122807        1.479639
```

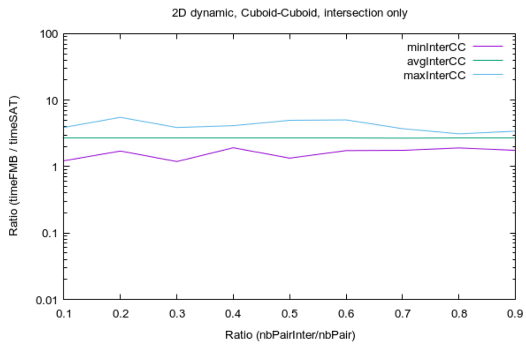
```
    13.880000       17200   32636   1.066079        1.494152        2.150000
            0.163265        1.128141        10.680000       0.163265
    1.274546        10.680000
0.5     74212   125780  1.016529        2.062993        4.977099
    0.075758        1.156487        14.111111       0.075758        1.609740
            14.111111       19988   29794   1.339768        2.674902
    4.977099        0.155556        1.148794        14.111111       0.155556
            1.911848        14.111111       18488   31664   1.226244
    1.999041        2.802920        0.138211        1.161943
    14.000000       0.138211        1.580492        14.000000       18508
    31502   1.418848        1.995142        2.608696        0.075758
    1.157796        13.625000       0.075758        1.576469
    13.625000       17228   32820   1.016529        1.494576        2.727273
            0.140187        1.156950        10.458333       0.140187
    1.325763        10.458333
0.6     74488   125504  0.954819        2.057871        5.046875
    0.132743        1.139723        14.037037       0.132743        1.690612
            14.037037       19814   29862   1.738693        2.673748
    5.046875        0.146789        1.096326        14.037037       0.146789
            2.042780        14.037037       18662   31396   1.371859
    1.997407        2.721973        0.146789        1.132185
    13.666667       0.146789        1.651318        13.666667       18460
    31316   1.195556        1.993735        3.835616        0.132743
    1.166805        13.541667       0.132743        1.662963
    13.541667       17552   32930   0.954819        1.494365        2.171779
            0.144231        1.160510        10.416667       0.144231
    1.360823        10.416667
0.7     74860   125128  1.161765        2.066658        3.723077
    0.131579        1.136448        14.260870       0.131579        1.787595
            14.260870       20306   29782   1.755000        2.676274
    3.723077        0.171717        1.111093        13.500000       0.171717
            2.206720        13.500000       18802   31164   1.406250
    1.999613        3.062937        0.131579        1.135340
    14.260870       0.131579        1.740331        14.260870       18656
    31402   1.334975        1.995057        3.293750        0.146789
    1.147764        13.913043       0.146789        1.740870
    13.913043       17096   32780   1.161765        1.494448        2.296970
            0.163043        1.149697        10.500000       0.163043
    1.391023        10.500000
0.8     74260   125732  0.743202        2.059377        3.426573
    0.146789        1.150253        13.750000       0.146789        1.877552
            13.750000       19724   30514   1.906077        2.675459
    3.108527        0.175258        1.148921        13.615385       0.175258
            2.370152        13.615385       18646   31356   1.108844
    1.998910        2.839416        0.146789        1.126765
    13.625000       0.146789        1.824481        13.625000       18506
    30944   0.825301        1.994680        3.426573        0.148148
    1.162588        13.750000       0.148148        1.828262
    13.750000       17384   32918   0.743202        1.494094        2.096386
            0.157895        1.162268        10.400000       0.157895
    1.427729        10.400000
0.9     74694   125302  1.012245        2.059103        3.406250
    0.145455        1.138906        14.041667       0.145455        1.967083
            14.041667       19902   30268   1.755000        2.674393
    3.406250        0.168675        1.133749        13.500000       0.168675
            2.520329        13.500000       18516   31040   1.183406
    1.997970        2.905405        0.145455        1.129149
    13.826087       0.145455        1.911088        13.826087       18738
    30898   1.281690        1.994435        3.401274        0.149533
    1.152957        14.041667       0.149533        1.910288
    14.041667       17538   33096   1.012245        1.494509        2.822086
            0.153846        1.139656        13.840000       0.153846
```

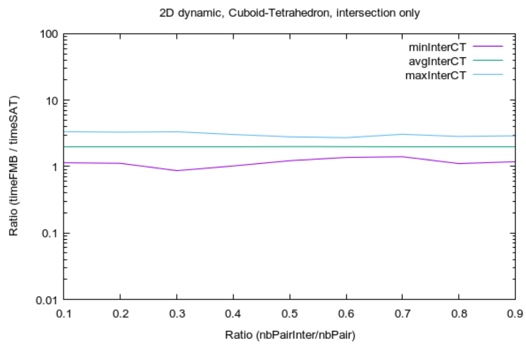
```
1.459024        13.840000
```

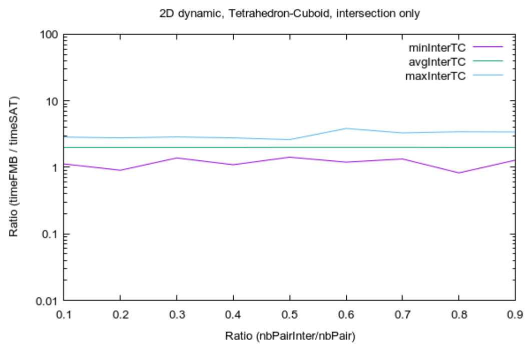


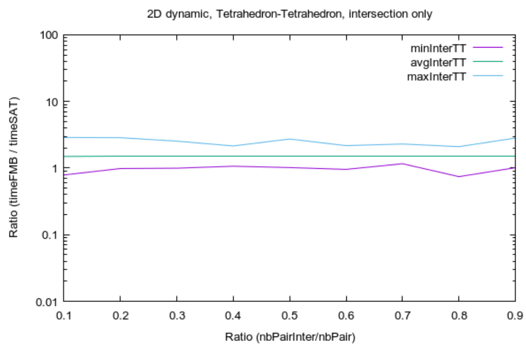

2D dynamic, Total, intersection only
minInter
avgInter
maxInter
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

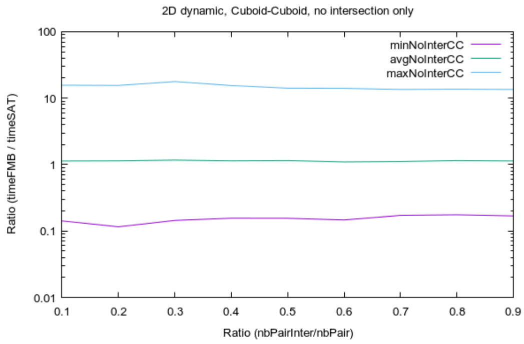

2D dynamic, Total, no intersection only
minNoInter
avgNoInter
maxNoInter
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

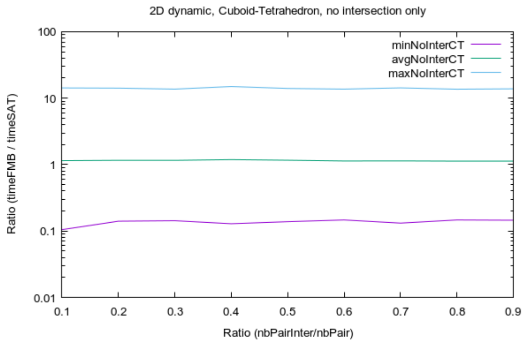

2D dynamic, Cuboid-Cuboid, intersection only
minInterCC
avgInterCC
maxInterCC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

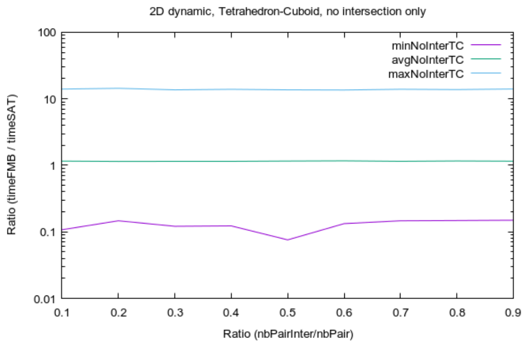

2D dynamic, Cuboid-Tetrahedron, intersection only
minInterCT
avgInterCT
maxInterCT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

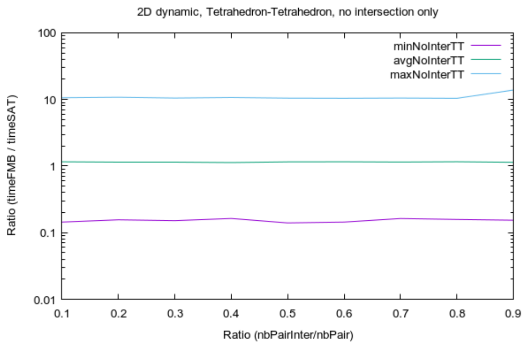

2D dynamic, Tetrahedron-Cuboid, intersection only
minInterTC
avgInterTC
maxInterTC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


2D dynamic, Tetrahedron-Tetrahedron, intersection only
minInterTT
avgInterTT
maxInterTT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


2D dynamic, Cuboid-Cuboid, no intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minNoInterCC
avgNoInterCC
maxNoInterCC
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, Cuboid-Tetrahedron, no intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minNoInterCT
avgNoInterCT
maxNoInterCT
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, Tetrahedron-Cuboid, no intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minNoInterTC
avgNoInterTC
maxNoInterTC
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

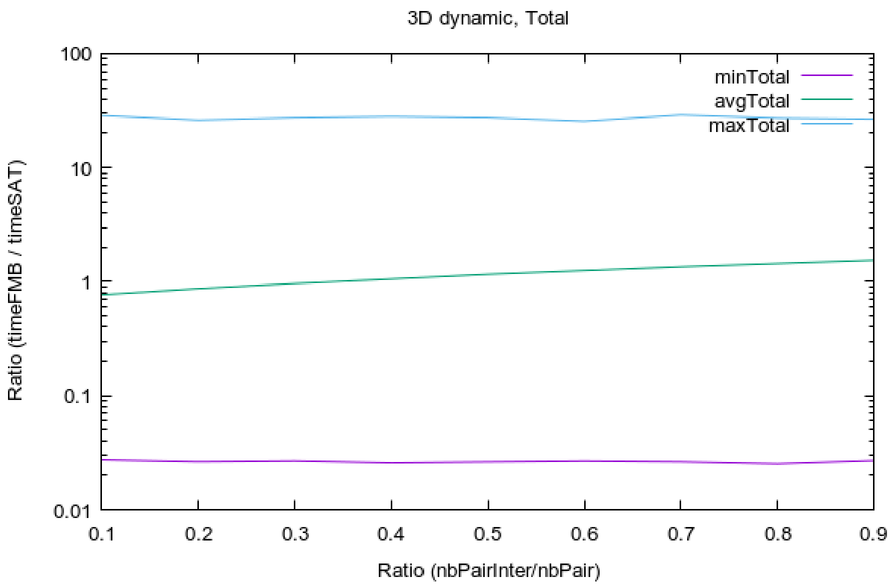

2D dynamic, Tetrahedron-Tetrahedron, no intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minNoInterTT
avgNoInterTT
maxNoInterTT
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

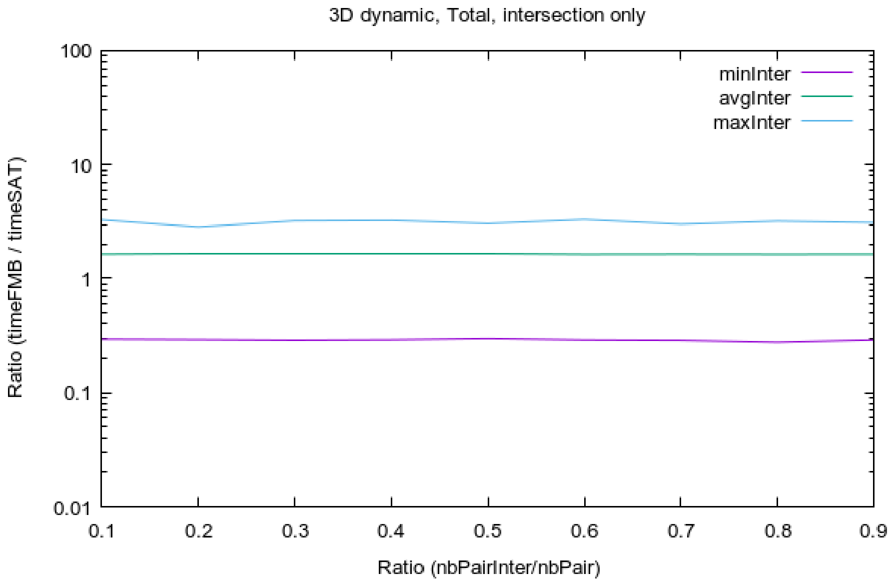
8.2.4 3D dynamic

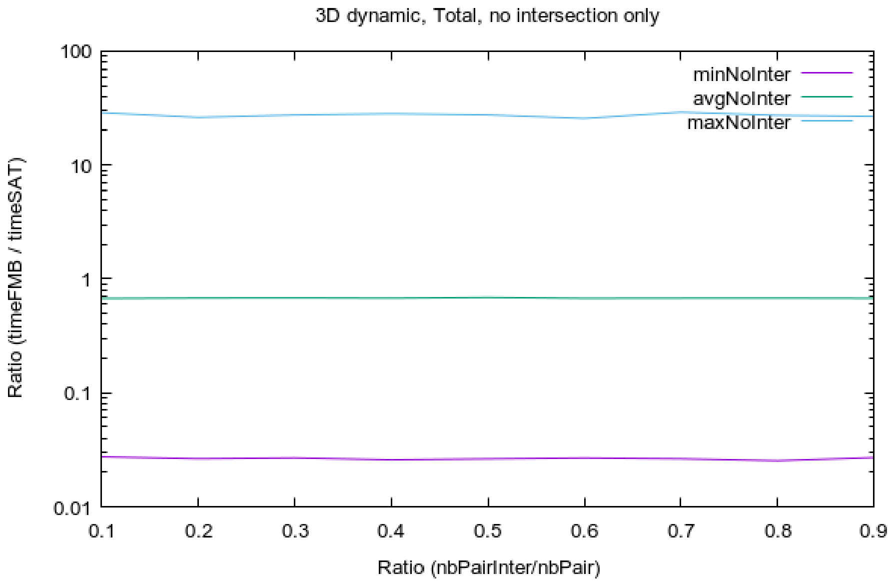
```
percPairInter   countInter      countNoInter    minInter        avgInter
            maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    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     52662   147338  0.295470        1.635678        3.287534
    0.027397        0.670604        28.750000       0.027397        0.767111
            28.750000       16028   33832   2.174427        2.649683
    3.287534        0.038988        0.605225        13.132743       0.038988
            0.809671        13.132743       13388   36738   0.470090
    1.425738        2.808812        0.029591        0.665893
    21.250000       0.029591        0.741878        21.250000       13246
    36650   0.475035        1.424402        2.806935        0.027397
    0.655610        21.240000       0.027397        0.732489
    21.240000       10000   40118   0.295470        0.571357        0.885655
            0.029160        0.743751        28.750000       0.029160
    0.726512        28.750000
0.2     52512   147488  0.293056        1.631850        2.843722
    0.026377        0.677084        26.162162       0.026377        0.868037
            26.162162       15998   33884   2.329609        2.649442
    2.843722        0.039911        0.584107        13.109649       0.039911
            0.997174        13.109649       13154   36370   0.465164
    1.422246        2.588880        0.027113        0.691414
    21.346154       0.027113        0.837580        21.346154       13208
    37282   0.462963        1.425412        2.626592        0.026459
    0.684160        22.140000       0.026459        0.832410
    22.140000       10152   39952   0.293056        0.568449        0.771102
            0.026377        0.736292        26.162162       0.026377
    0.702724        26.162162
0.3     52496   147504  0.289132        1.630786        3.222222
    0.026814        0.679176        27.447368       0.026814        0.964659
            27.447368       15936   34158   1.953381        2.647826
    3.222222        0.039785        0.597588        13.190265       0.039785
            1.212660        13.190265       13194   36684   0.464770
    1.424302        2.741058        0.028357        0.680291
    22.306122       0.028357        0.903494        22.306122       13252
    37116   0.448023        1.422098        2.767315        0.027597
    0.681975        22.100000       0.027597        0.904012
    22.100000       10114   39546   0.289132        0.571102        0.770115
            0.026814        0.745986        27.447368       0.026814
    0.693521        27.447368
0.4     52462   147538  0.292103        1.628984        3.253610
    0.025777        0.675281        28.166667       0.025777        1.056763
            28.166667       15850   33736   2.188848        2.648731
    3.253610        0.039046        0.585627        13.151111       0.039046
            1.410869        13.151111       13170   37162   0.465327
    1.424639        2.745283        0.025777        0.682961
    21.764706       0.025777        0.979632        21.764706       13304
    37032   0.464484        1.424049        2.589540        0.027486
    0.674866        22.693878       0.027486        0.974539
```

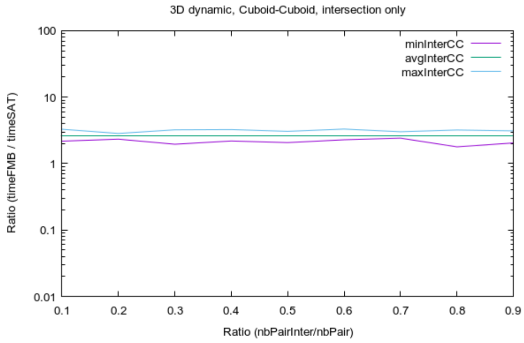
22.693878 10138 39608 0.292103 0.569080 0.944763
0.026295 0.744827 28.166667 0.026295
0.674528 28.166667
0.5 52522 147476 0.299363 1.629522 3.055505
0.026295 0.685067 27.486486 0.026295 1.157294
27.486486 15856 33774 2.067664 2.649247
3.055505 0.039046 0.623731 13.225532 0.039046
1.636489 13.225532 13282 36778 0.432852
1.427203 2.567460 0.027892 0.682456
22.160000 0.027892 1.054829 22.160000 13214
36988 0.453205 1.424185 2.895899 0.026295
0.677340 22.183673 0.026295 1.050762
22.183673 10170 39936 0.299363 0.570700 0.780714
0.026480 0.746499 27.486486 0.026480
0.658599 27.486486
0.6 52412 147588 0.291034 1.631692 3.304867
0.026772 0.672917 25.578947 0.026772 1.248182
25.578947 15944 33724 2.279595 2.649855
3.304867 0.039779 0.608497 13.133333 0.039779
1.833312 13.133333 13216 37232 0.465226
1.425832 2.593873 0.027575 0.667868
22.160000 0.027575 1.122646 22.160000 13120
36776 0.472626 1.421505 2.613994 0.027620
0.663361 20.425926 0.027620 1.118247
20.425926 10132 39856 0.291034 0.570176 0.965982
0.026772 0.740960 25.578947 0.026772
0.638490 25.578947
0.7 52102 147898 0.288171 1.636444 3.011883
0.026336 0.674717 29.055556 0.026336 1.347926
29.055556 15938 34070 2.413223 2.649690
3.011883 0.039779 0.599181 13.109649 0.039779
2.034537 13.109649 13180 36894 0.463415
1.425449 2.598619 0.027938 0.678214
21.346939 0.027938 1.201278 21.346939 13022
36934 0.469331 1.424588 2.573333 0.027419
0.669035 20.420000 0.027419 1.197922
20.420000 9962 40000 0.288171 0.571458 0.918613
0.026336 0.741077 29.055556 0.026336
0.622344 29.055556
0.8 52046 147954 0.277704 1.630316 3.201465
0.025307 0.675822 27.250000 0.025307 1.439417
27.250000 15864 34166 1.784398 2.649216
3.201465 0.039691 0.588428 13.105727 0.039691
2.237058 13.105727 12904 36702 0.467488
1.426547 2.630487 0.026877 0.684138
21.509804 0.026877 1.278065 21.509804 13094
36994 0.466200 1.422146 2.577566 0.028878
0.666454 21.700000 0.028878 1.271008
21.700000 10184 40092 0.277704 0.568985 0.834207
0.025307 0.751329 27.250000 0.025307
0.605454 27.250000
0.9 52724 147276 0.290947 1.632746 3.107468
0.026984 0.672882 26.675676 0.026984 1.536759
26.675676 16018 33676 2.044050 2.650071
3.107468 0.038380 0.612551 13.146667 0.038380
2.446319 13.146667 13274 37156 0.477560
1.425162 2.569841 0.027755 0.666932
21.081633 0.027755 1.349339 21.081633 13298
36586 0.473038 1.425413 2.565696 0.027642
0.658084 21.408163 0.027642 1.348680
21.408163 10134 39858 0.290947 0.568711 0.776288
0.026984 0.742984 26.675676 0.026984

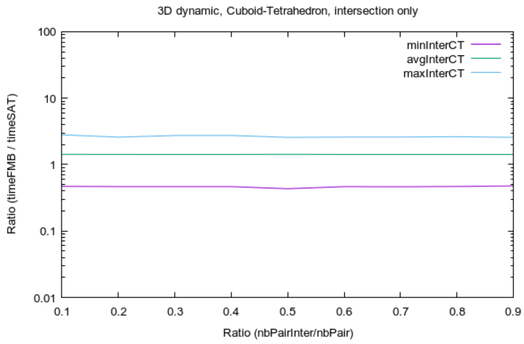
0.586138 26.675676

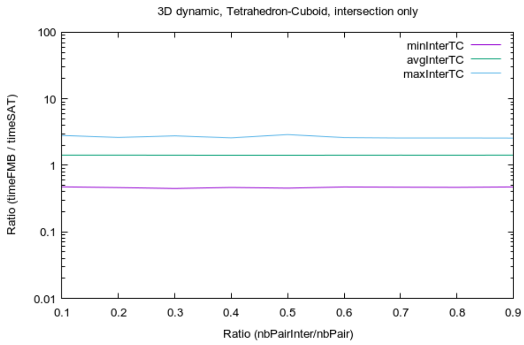


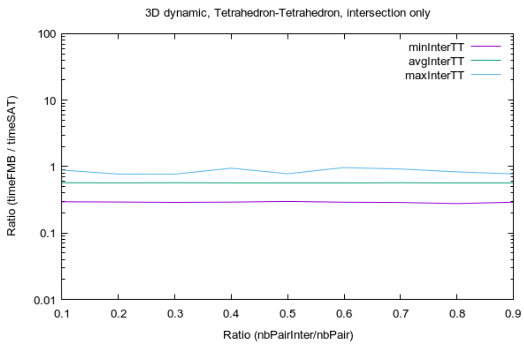

3D dynamic, Total, intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minInter
avgInter
maxInter
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

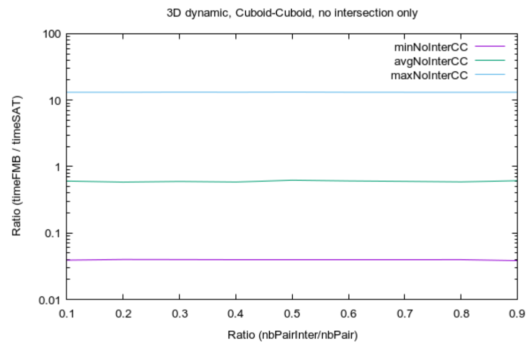

3D dynamic, Total, no intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minNoInter
avgNoInter
maxNoInter
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

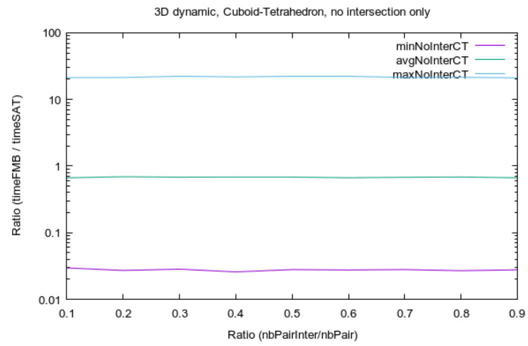

3D dynamic, Cuboid-Cuboid, intersection only
minInterCC
avgInterCC
maxInterCC
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

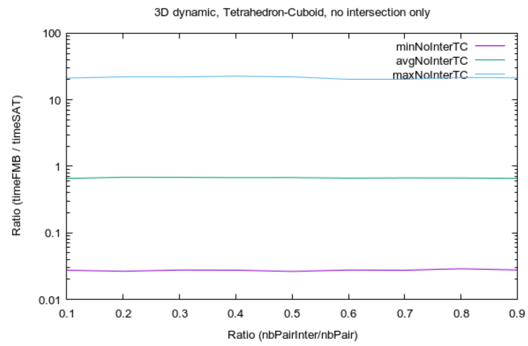

3D dynamic, Cuboid-Tetrahedron, intersection only
minInterCT
avgInterCT
maxInterCT
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

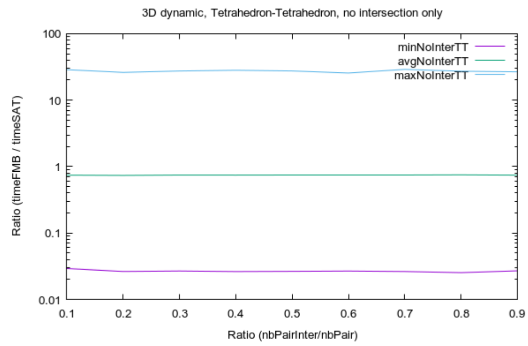

3D dynamic, Tetrahedron-Cuboid, intersection only
minInterTC
avgInterTC
maxInterTC
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


3D dynamic, Tetrahedron-Tetrahedron, intersection only
minInterTT
avgInterTT
maxInterTT
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 (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


3D dynamic, Cuboid-Cuboid, no intersection only
minNoInterCC
avgNoInterCC
maxNoInterCC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


3D dynamic, Cuboid-Tetrahedron, no intersection only
minNoInterCT
avgNoInterCT
maxNoInterCT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


3D dynamic, Tetrahedron-Cuboid, no intersection only
minNoInterTC
avgNoInterTC
maxNoInterTC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


3D dynamic, Tetrahedron-Tetrahedron, no intersection only
minNoInterTT
avgNoInterTT
maxNoInterTT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

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 algorithms as the tests they performed are completely different. But in average, the FMB algorithm has the advantage for all but the 2D dynamic case.

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_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              12
On-line CPU(s) list: 0-11
Thread(s) per core:  2
Core(s) per socket:  6
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               158
Model name:          Intel(R) Core(TM) i7-8700T CPU @ 2.40GHz
Stepping:            10
CPU MHz:             2216.548
CPU max MHz:         4000.0000
```

```
CPU min MHz:          800.0000
BogoMIPS:             4800.00
Virtualization:       VT-x
L1d cache:            32K
L1i cache:            32K
L2 cache:             256K
L3 cache:             12288K
NUMA node0 CPU(s):    0-11
Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
    mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
    syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
    rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni
    pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr
     pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes
    xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb
    invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept
     vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid
    rtm mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1
    xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp
    md_clear flush_l1d

=============

gcc -v

Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.4.0-1
    ubuntu1~18.04.1' --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_64-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_64-linux-gnu
     --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)
```

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 for 3D Frames
bool CheckAxis3D(
  const Frame3D* const that,
  const Frame3D* const tho,
  const double* const axis);

// Check the intersection constraint along one axis for moving 3D Frames
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. It also includes command used to run the unit tests, validation and qualification, and to generate the documentation.

```
COMPILER=gcc
OPTIMIZATION=-O3

all : compile run plot getRuntimeEnvironment 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 -

getRuntimeEnvironment:
        echo "uname -v\n" > runtimeEnv.txt; uname -v >> runtimeEnv.txt; echo
             "\n============\n" >> runtimeEnv.txt; echo "lshw -short\n" >>
            runtimeEnv.txt; sudo lshw -short >> runtimeEnv.txt; echo "\n
```

```
    ============\n" >> runtimeEnv.txt; echo "lscpu\n" >> runtimeEnv
    .txt; lscpu >> runtimeEnv.txt; echo "\n============\n" >>
    runtimeEnv.txt; echo "gcc -v\n" >> runtimeEnv.txt; gcc -v 1>>
    runtimeEnv.txt 2>> runtimeEnv.txt
```

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
```

10.3.5 Doc

```
latex:
        pdflatex -synctex=1 -interaction=nonstopmode -shell-escape  fmb.tex
```

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.