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

P. Baillehache

December 31, 2019

Abstract

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

Contents

1 Notations

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

2 Definition of the problem

2.1 Static case

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and for a tetrahedron:

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

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

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

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

2.2 Dynamic case

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

If A and B are two cuboids:

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

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

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

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

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

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

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

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

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

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

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

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

and for a tetrahedron:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3 Solution

3.1 Fourier-Motzkin elimination method

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

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

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

with

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

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

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

where

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

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

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

and

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

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

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

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

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

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

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

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

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

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

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

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

4 Algorithms

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

4.1 Frames

algo

4.2 2D static

```
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
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
  nbRemainRows = 0

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

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

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

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

      // If coefficients of the eliminated variable in the two rows have
      // dIFferent signs and are not null
      IF (sgnMIRowIVar <> sgn(M[jRow][iVar]) AND
          fabsMIRowIVar > EPSILON AND
          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[nbRemainRows][jCol] =
              M[iRow][iCol] / fabsMIRowIVar +
              M[jRow][iCol] / fabs(M[jRow][iVar])

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

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

          END

        END

        // Update the right side of the inequality
        Yp[nbRemainRows] =
          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[nbRemainRows] < sumNegCoeff - EPSILON) {

          // Given that X is in [0,1], the system is inconsistent
//printf("inconsistent %.9f %.9f\n",Yp[nbRemainRows], sumNegCoeff + EPSILON)
          RETURN TRUE

        END

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

      END

    END

  END

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

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

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

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

      // Shortcut
      double* MpnbRemainRows = Mp[nbRemainRows]

      // Copy this row into the result system excluding the eliminated
      // variable
      FOR (int iCol = 0, jCol = 0
```

```
           iCol < nbCols
           ++iCol) {

        IF (iCol <> iVar) {

          MpnbRemainRows[jCol] = MiRow[iCol]

          ++jCol

        END

      END

      Yp[nbRemainRows] = Y[iRow]

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

    END

  END

  // If we reach here the system is not inconsistent
  RETURN FALSE

END

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

  // Shortcuts
  double* min = bdgBox.min + iVar
  double* max = bdgBox.max + iVar

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

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

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

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

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

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

        // Update the maximum bound
        *max = y

      END

    // 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

      END

    END

  END

END

// Test FOR intersection between Frame 'that' and Frame 'tho'
// Return TRUE IF the two Frames are intersecting, ELSE FALSE
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', ELSE 'bdgBox' is not modIFied
// If 'bdgBox' is null, the result AABB is not memorized (to use IF
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be dIFferent
// of the resulting AABB of FMBTestIntersection(B,A)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection2D(
  const Frame2D* const that,
  const Frame2D* const tho,
        AABB2D* const bdgBox) {
//Frame2DPrint(that) printf("\n")
//Frame2DPrint(tho) printf("\n")
  // Get the projection of the Frame 'tho' in Frame 'that' coordinates
  // system
  Frame2D thoProj
  Frame2DImportFrame(that, tho, &thoProj)

  // Declare two variables to memorize the system to be solved M.X <= Y
  // (M arrangement is [iRow][iCol])
  double M[8][2]
  double Y[8]

  // Create the inequality system

  // -sum_iC_j,iX_i<=O_j
  M[0][0] = -thoProj.comp[0][0]
```

```
M[0][1] = -thoProj.comp[1][0]
Y[0] = thoProj.orig[0]
IF (Y[0] < neg(M[0][0]) + neg(M[0][1]))
  RETURN FALSE

M[1][0] = -thoProj.comp[0][1]
M[1][1] = -thoProj.comp[1][1]
Y[1] = thoProj.orig[1]
IF (Y[1] < neg(M[1][0]) + neg(M[1][1]))
  RETURN FALSE

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

M[3][0] = 0.0
M[3][1] = -1.0
Y[3] = 0.0

// Variable to memorise the nb of rows in the system
int nbRows = 4

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

END

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

  END

  // Solve the system

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

  // Declare variables to eliminate the first variable
  double Mp[16][2]
  double Yp[16]
  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
//printf("inconsisten A\n")
    RETURN FALSE

  END

  // 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
//printf("bound %f %f\n",bdgBoxLocal.min[SND_VAR],bdgBoxLocal.max[SND_VAR])
    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
//printf("inter\n")
    RETURN TRUE

  END

  // 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)
//printf("inconsistent B %d\n",inconsistency)
  // Get the bounds FOR the remaining first variable
  GetBound2D(
    FST_VAR,
    Mp,
    Yp,
    nbRowsP,
    &bdgBoxLocal)
//printf("bound %f %f\n",bdgBoxLocal.min[FST_VAR],bdgBoxLocal.max[FST_VAR])

  // If the user requested the resulting bounding box
  IF (bdgBox <> NULL) {

    // Memorize the result
    *bdgBox = bdgBoxLocal

  END

  // If we've reached here the two Frames are intersecting
//printf("inter\n")
  RETURN TRUE

END
```

4.3 3D static

```
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
EPSILON = 0.0000001

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

FUNCTION GetBound3D(iVar, M, Y, nbRows, bdgBox)
  bdgBox.min[iVar] = 0.0
  bdgBox.max[iVar] = 1.0
```

```
  FOR jRow = 0..(nbRows - 1)
    IF M[jRow][0] > EPSILON
      y = Y[jRow] / M[jRow][0]
      IF bdgBox.max[iVar] > y
        bdgBox.max[iVar] = y
      END IF
    ELSE IF M[jRow][0] < -EPSILON
      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
  M[3][0] = -1.0
  M[3][1] = 0.0
  M[3][2] = 0.0
  Y[3] = 0.0
  M[4][0] = 0.0
  M[4][1] = -1.0
  M[4][2] = 0.0
  Y[4] = 0.0
  M[5][0] = 0.0
  M[5][1] = 0.0
  M[5][2] = -1.0
  Y[5] = 0.0
  nbRows = 6
  IF that.type == FrameCuboid
    M[nbRows][0] = thoProj.comp[0][0]
    M[nbRows][1] = thoProj.comp[1][0]
    M[nbRows][2] = thoProj.comp[2][0]
    Y[nbRows] = 1.0 - thoProj.orig[0]
    IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                   neg(M[nbRows][2])
      RETURN FALSE
    END IF
    nbRows = nbRows + 1
    M[nbRows][0] = thoProj.comp[0][1]
```

```
  M[nbRows][1] = thoProj.comp[1][1]
  M[nbRows][2] = thoProj.comp[2][1]
  Y[nbRows] = 1.0 - thoProj.orig[1]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
  M[nbRows][0] = thoProj.comp[0][2]
  M[nbRows][1] = thoProj.comp[1][2]
  M[nbRows][2] = thoProj.comp[2][2]
  Y[nbRows] = 1.0 - thoProj.orig[2]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
ELSE
  M[nbRows][0] =
    thoProj.comp[0][0] + thoProj.comp[0][1] + thoProj.comp[0][2]
  M[nbRows][1] =
    thoProj.comp[1][0] + thoProj.comp[1][1] + thoProj.comp[1][2]
  M[nbRows][2] =
    thoProj.comp[2][0] + thoProj.comp[2][1] + thoProj.comp[2][2]
  Y[nbRows] =
    1.0 - thoProj.orig[0] - thoProj.orig[1] - thoProj.orig[2]
  IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
                 neg(M[nbRows][2])
    RETURN FALSE
  END IF
  nbRows = nbRows + 1
END
IF tho.type == FrameCuboid {
  M[nbRows][0] = 1.0
  M[nbRows][1] = 0.0
  M[nbRows][2] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = 1.0
  M[nbRows][2] = 0.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
  M[nbRows][0] = 0.0
  M[nbRows][1] = 0.0
  M[nbRows][2] = 1.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
ELSE
  M[nbRows][0] = 1.0
  M[nbRows][1] = 1.0
  M[nbRows][2] = 1.0
  Y[nbRows] = 1.0
  nbRows = nbRows + 1
END
inconsistency =
  ElimVar3D(FST_VAR, M, Y, nbRows, 3, Mp, Yp, nbRowsP)
IF inconsistency == TRUE
  RETURN FALSE
END
inconsistency =
  ElimVar3D(FST_VAR, Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
```

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

origP3D = [0.0, 0.0, 0.0]
compP3D = [
  [1.0, 0.0, 0.0],
  [0.0, 1.0, 0.0],
  [0.0, 0.0, 1.0]]
P3D = Frame3DCreateStatic(FrameTetrahedron, origP3D, compP3D)
origQ3D = [0.5, 0.5, 0.5]
compQ3D = [
  [2.0, 0.0, 0.0],
  [0.0, 2.0, 0.0],
  [0.0, 0.0, 2.0]]
Q3D = Frame3DCreateStatic(FrameTetrahedron, origQ3D, compQ3D)
isIntersecting3D = FMBTestIntersection3D(P3D, Q3D, bdgBox3DLocal)
IF isIntersecting3D == TRUE
  PRINT "Intersection detected in AABB "
  Frame3DExportBdgBox(Q3D, bdgBox3DLocal, bdgBox3D)
  AABB3DPrint(bdgBox3D)
ELSE
  PRINT "No intersection."
END IF
```

4.4 2D dynamic

```
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
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
  nbRemainRows = 0

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

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

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

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

    // If coefficients of the eliminated variable in the two rows have
    // dIFferent signs and are not null
    IF (sgnMIRowIVar <> sgn(M[jRow][iVar]) AND
        fabsMIRowIVar > EPSILON AND
        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[nbRemainRows][jCol] =
            M[iRow][iCol] / fabsMIRowIVar +
            M[jRow][iCol] / fabs(M[jRow][iVar])

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

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

        END

      END

      // Update the right side of the inequality
      Yp[nbRemainRows] =
        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[nbRemainRows] < sumNegCoeff - EPSILON) {

        // Given that X is in [0,1], the system is inconsistent
        RETURN TRUE
```

```
        END

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

      END

    END

  END

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

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

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

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

      // Shortcut
      double* MpnbRemainRows = Mp[nbRemainRows]

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

        IF (iCol <> iVar) {

          MpnbRemainRows[jCol] = MiRow[iCol]

          ++jCol

        END

      END

      Yp[nbRemainRows] = Y[iRow]

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

    END

  END

  // If we reach here the system is not inconsistent
  RETURN FALSE

END

// 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

      END

    // 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

      END

    END

  END
```

```
END

// 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

  M[2][0] = -1.0
  M[2][1] = 0.0
  M[2][2] = 0.0
  Y[2] = 0.0

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

  M[4][0] = 0.0
  M[4][1] = 0.0
  M[4][2] = 1.0
  Y[4] = 1.0

  M[5][0] = 0.0
  M[5][1] = 0.0
```

```
M[5][2] = -1.0
Y[5] = 0.0

// Variable to memorise the nb of rows in the system
int nbRows = 6

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

END

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

END

// Solve the system

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

// Declare variables to eliminate the first variable
double Mp[25][3]
double Yp[25]
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

END

// Declare variables to eliminate the second variable
double Mpp[169][3]
double Ypp[169]
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

END

// Get the bounds FOR the remaining third variable
```

```
GetBound2DTime(
  THD_VAR,
  Mpp,
  Ypp,
  nbRowsPP,
  &bdgBoxLocal)

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

  // The two Frames are not in intersection
  RETURN FALSE

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

  // Immediately RETURN TRUE
  RETURN TRUE

END

// 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

  END

  // If we've reached here the two Frames are intersecting
  RETURN TRUE

END
```

4.5 3D dynamic

```
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
EPSILON = 0.0000001

FUNCTION ElimVar3DTime(iVar, M, Y, nbRows, nbCols, Mp, Yp, nbRemainRows)
  nbRemainRows = 0
  FOR iRow = 0..(nbRows - 2)
```

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

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

FUNCTION FMBTestIntersection3DTime(that, tho, bdgBox)
  Frame3DTimeImportFrame(that, tho, thoProj)
  M[0][0] = -thoProj.comp[0][0]
```

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

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

```
  Y[nbRows] = 0.0
  nbRows = nbRows + 1
  inconsistency =
    ElimVar3DTime(FST_VAR, M, Y, nbRows, 4, Mp, Yp, nbRowsP)
  IF inconsistency == TRUE
    RETURN FALSE
  END IF
  inconsistency =
    ElimVar3DTime(FST_VAR, Mp, Yp, nbRowsP, 3, Mpp, Ypp, nbRowsPP)
  IF inconsistency == TRUE
    RETURN FALSE
  END IF
  inconsistency =
    ElimVar3DTime(FST_VAR, Mpp, Ypp, nbRowsPP, 2, Mppp, Yppp, nbRowsPPP)
  IF inconsistency == TRUE
    RETURN FALSE
  END IF
  GetBound3DTime(FOR_VAR, Mppp, Yppp, nbRowsPPP, bdgBoxLocal)
  IF bdgBoxLocal.min[FOR_VAR] >= bdgBoxLocal.max[FOR_VAR]
    RETURN FALSE
  END IF
  ElimVar3DTime(SND_VAR, Mpp, Ypp, nbRowsPP, 2, Mppp, Yppp, nbRowsPPP)
  GetBound3DTime(THD_VAR, Mppp, Yppp, nbRowsPPP, bdgBoxLocal)
  ElimVar3DTime(FOR_VAR, M, Y, nbRows, 4, Mp, Yp, nbRowsP)
  ElimVar3DTime(THD_VAR, Mp, Yp, nbRowsP, 3, Mpp, Ypp, nbRowsPP)
  ElimVar3DTime(SND_VAR, Mpp, Ypp, nbRowsPP, 2, Mppp, Yppp, nbRowsPPP)
  GetBound3DTime(FST_VAR, Mppp, Yppp, nbRowsPPP, bdgBoxLocal)
  ElimVar3DTime(FST_VAR, Mpp, Ypp, nbRowsPP, 2, Mppp, Yppp, nbRowsPPP)
  GetBound3DTime(SND_VAR, Mppp, Yppp, nbRowsPPP, bdgBoxLocal)
  bdgBox = bdgBoxLocal
  RETURN TRUE
END FUNCTION

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

5 Implementation

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

5.1 Frames

5.1.1 Header

```
#ifndef __FRAME_H_
#define __FRAME_H_

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

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

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

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

typedef enum {
  FrameCuboid,
  FrameTetrahedron
} FrameType;

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

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

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

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

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

// Axis unaligned cuboid and tetrahedron structure
typedef struct {
```

```
  FrameType type;
  double orig[2];
  double comp[2][2];
  // AABB of the frame
  AABB2D bdgBox;
  // Inverted components used during computation
  double invComp[2][2];
} Frame2D;

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

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

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

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

// Print the AABB 'that' on stdout
// Output format is
// (min[0], min[1], min[2], min[3])-(max[0], max[1], max[2], max[3])
void AABB2DPrint(const AABB2D* const that);
void AABB3DPrint(const AABB3D* const that);
void AABB2DTimePrint(const AABB2DTime* const that);
void AABB3DTimePrint(const AABB3DTime* const that);

// Print the Frame 'that' on stdout
// Output format is
// (orig[0], orig[1], orig[2])
// (comp[0][0], comp[0][1], comp[0][2])
// (comp[1][0], comp[1][1], comp[1][2])
// (comp[2][0], comp[2][1], comp[2][2])
// (speed[0], speed[1], speed[2])
void Frame2DPrint(const Frame2D* const that);
void Frame3DPrint(const Frame3D* const that);
void Frame2DTimePrint(const Frame2DTime* const that);
void Frame3DTimePrint(const Frame3DTime* const that);
```

```
// Create a static Frame structure of FrameType 'type',
// at position 'orig' with components 'comp' ([iComp][iAxis])
Frame2D Frame2DCreateStatic(
  const FrameType type,
     const double orig[2],
     const double comp[2][2]);
Frame3D Frame3DCreateStatic(
  const FrameType type,
     const double orig[3],
     const double comp[3][3]);
Frame2DTime Frame2DTimeCreateStatic(
  const FrameType type,
     const double orig[2],
     const double speed[2],
     const double comp[2][2]);
Frame3DTime Frame3DTimeCreateStatic(
  const FrameType type,
     const double orig[3],
     const double speed[3],
     const double comp[3][3]);

// Project the Frame 'Q' in the Frame 'P' 's coordinates system and
// memorize the result in the Frame 'Qp'
void Frame2DImportFrame(
  const Frame2D* const P,
  const Frame2D* const Q,
        Frame2D* const Qp);
void Frame3DImportFrame(
  const Frame3D* const P,
  const Frame3D* const Q,
        Frame3D* const Qp);
void Frame2DTimeImportFrame(
  const Frame2DTime* const P,
  const Frame2DTime* const Q,
        Frame2DTime* const Qp);
void Frame3DTimeImportFrame(
  const Frame3DTime* const P,
  const Frame3DTime* const Q,
        Frame3DTime* const Qp);

// Export the AABB 'bdgBox' from 'that' 's coordinates system to
// the real coordinates system and update 'bdgBox' with the resulting
// AABB
void Frame2DExportBdgBox(
  const Frame2D* const that,
   const AABB2D* const bdgBox,
         AABB2D* const bdgBoxProj);
void Frame3DExportBdgBox(
  const Frame3D* const that,
   const AABB3D* const bdgBox,
         AABB3D* const bdgBoxProj);
void Frame2DTimeExportBdgBox(
  const Frame2DTime* const that,
   const AABB2DTime* const bdgBox,
         AABB2DTime* const bdgBoxProj);
void Frame3DTimeExportBdgBox(
  const Frame3DTime* const that,
   const AABB3DTime* const bdgBox,
         AABB3DTime* const bdgBoxProj);

// Power function for integer base and exponent
```

```
// Return 'base' ^ 'exp'
int powi(
           int base,
  unsigned int exp);

#endif
```

5.1.2 Body

```
#include "frame.h"

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

#define EPSILON 0.0000001

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

// Update the inverse components of the Frame 'that'
void Frame2DUpdateInv(Frame2D* const that);
void Frame3DUpdateInv(Frame3D* const that);
void Frame2DTimeUpdateInv(Frame2DTime* const that);
void Frame3DTimeUpdateInv(Frame3DTime* const that);

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

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

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

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

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

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

    }

  }

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

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

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

      if (that.type == FrameCuboid) {
```

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

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

        }

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

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

        }

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

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

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

        }

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

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

        }

      }

    }

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

  }


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

  // Return the new Frame
  return that;

}

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

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

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

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

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

  }

}

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

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

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

    if (that.type == FrameCuboid) {

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

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

      }

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

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

      }

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

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

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

      }

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

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

      }

    }

  }

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

}


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

// Return the new Frame
return that;
```

```
}

Frame2DTime Frame2DTimeCreateStatic(
  const FrameType type,
     const double orig[2],
     const double speed[2],
     const double comp[2][2]) {

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

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

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

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

    }

  }

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

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

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

      if (that.type == FrameCuboid) {

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

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

        }

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

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

        }

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

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

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

        }

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

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

        }

      }

    }

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

      min += that.speed[iAxis];

    }

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

      max += that.speed[iAxis];

    }

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

  }

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


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

  // Return the new Frame
  return that;

}

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

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

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

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

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

    }

  }
```

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

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

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

    if (that.type == FrameCuboid) {

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

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

      }

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

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

      }

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

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

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

      }

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

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

      }

    }

  }

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

    min += that.speed[iAxis];

  }

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

    max += that.speed[iAxis];

  }

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

}

that.bdgBox.min[3] = 0.0;
```

```
  that.bdgBox.max[3] = 1.0;

  // Calculate the inverse matrix
  Frame3DTimeUpdateInv(&that);

  // Return the new Frame
  return that;

}

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

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

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

  tic[0][0] = tc[1][1] / det;
  tic[0][1] = -tc[0][1] / det;
  tic[1][0] = -tc[1][0] / det;
  tic[1][1] = tc[0][0] / det;

}

void Frame3DUpdateInv(Frame3D* const that) {

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

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

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

}

// Update the inverse components of the Frame 'that'
void Frame2DTimeUpdateInv(Frame2DTime* const that) {
```

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

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

  tic[0][0] = tc[1][1] / det;
  tic[0][1] = -tc[0][1] / det;
  tic[1][0] = -tc[1][0] / det;
  tic[1][1] = tc[0][0] / det;

}

void Frame3DTimeUpdateInv(Frame3DTime* const that) {

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

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

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

}

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

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

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

```
  const double  (*qc)[2] = Q->comp;

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

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

  }

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

    qpo[i] = 0.0;

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

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

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

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

      }
    }
  }
}

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

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

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

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

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

  }

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

    qpo[i] = 0.0;

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

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

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

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

      }
    }
  }
}

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

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

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

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

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

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

  }

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

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

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

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

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

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

      }
```

```
    }
  }
}

void Frame3DTimeImportFrame(
  const Frame3DTime* const P,
  const Frame3DTime* const Q,
        Frame3DTime* const Qp) {

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

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

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

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

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

  }

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

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

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

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

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

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

      }
    }
  }
}

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

```
       AABB2D* const bdgBoxProj) {

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

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

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

  bbpma[i] = to[i];

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

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

  }

  bbpmi[i] = bbpma[i];

}

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

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

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

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

  }

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

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

    w[i] = to[i];

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

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

```
      }
    }

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

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

        bbpmi[i] = w[i];

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

        bbpma[i] = w[i];

      }
    }
  }

}

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

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

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

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

    bbpma[i] = to[i];

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

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

    }

    bbpmi[i] = bbpma[i];

  }

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

    // Declare a variable to memorize the coordinates of the vertex in
```

```
    // 'that' 's coordinates system
    double v[3];

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

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

    }

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

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

      w[i] = to[i];

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

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

      }
    }

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

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

        bbpmi[i] = w[i];

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

        bbpma[i] = w[i];

      }
    }
  }

}

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

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

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

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

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

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

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

  }

  bbpmi[i] = bbpma[i];

}

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

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

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

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

  }

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

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

    w[i] = to[i];

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

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

    }
  }

  // Update the coordinates of the result AABB
  for (int i = 2;
```

```
        i--;) {

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

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

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

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

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

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

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

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

      }
    }
  }

}

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

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

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

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

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

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

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

    }

    bbpmi[i] = bbpma[i];

  }
```

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

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

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

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

  }

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

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

    w[i] = to[i];

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

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

    }
  }

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

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

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

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

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

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

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

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

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

    }
```

```
    }
  }

}

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

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

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

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

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

  }
  printf(")");

}

void AABB3DPrint(const AABB3D* const that) {

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

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

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

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

  }
  printf(")");

}

void AABB2DTimePrint(const AABB2DTime* const that) {

  printf("minXYT(");
```

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

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

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

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

  }
  printf(")");

}

void AABB3DTimePrint(const AABB3DTime* const that) {

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

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

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

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

  }
  printf(")");

}

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

```
       i < 2;
       ++i) {

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

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

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

    }
  }
  printf(")");

}

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

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

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

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

    }
  }
  printf(")");

}
```

```
void Frame2DTimePrint(const Frame2DTime* const that) {
  if (that->type == FrameTetrahedron) {
    printf("T");
  } else if (that->type == FrameCuboid) {
    printf("C");
  }
  printf("o(");
  for (int i = 0;
       i < 2;
       ++i) {

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

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

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

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

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

    }
  }
  printf(")");

}

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

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

  }
  printf(") s(");
```

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

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

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

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

    }
  }
  printf(")");

}

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

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

      res *= base;

    }
    return res;
}
```

5.2 FMB

5.2.1 2D static

Header

```
#ifndef __FMB2D_H_
#define __FMB2D_H_

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

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

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
```

```
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
// 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
  *nbRemainRows = 0;

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

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

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

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

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

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

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

          if (iCol != iVar) {
```

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

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

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

        }

      }

      // Update the right side of the inequality
      Yp[*nbRemainRows] =
        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[*nbRemainRows] < sumNegCoeff - EPSILON) {

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

      }

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

    }

  }

}

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

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

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

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

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

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

```
        if (iCol != iVar) {

          MpnbRemainRows[jCol] = MiRow[iCol];

          ++jCol;

        }

      }

      Yp[*nbRemainRows] = Y[iRow];

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

    }

  }

  // If we reach here the system is not inconsistent
  return false;

}

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

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

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

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

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

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

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

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

        // Update the maximum bound
        *max = y;

      }

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

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

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

        // Update the minimum bound
        *min = y;

      }

    }

  }

}

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection2D(
  const Frame2D* const that,
  const Frame2D* const tho,
        AABB2D* const bdgBox) {
//Frame2DPrint(that);printf("\n");
//Frame2DPrint(tho);printf("\n");
  // Get the projection of the Frame 'tho' in Frame 'that' coordinates
  // system
  Frame2D thoProj;
  Frame2DImportFrame(that, tho, &thoProj);

  // Declare two variables to memorize the system to be solved M.X <= Y
  // (M arrangement is [iRow][iCol])
  double M[8][2];
  double Y[8];

  // Create the inequality system

  // -sum_iC_j,iX_i<=O_j
  M[0][0] = -thoProj.comp[0][0];
  M[0][1] = -thoProj.comp[1][0];
  Y[0] = thoProj.orig[0];
```

```
if (Y[0] < neg(M[0][0]) + neg(M[0][1]))
  return false;

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

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

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

// Variable to memorise the nb of rows in the system
int nbRows = 4;

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;

}

// Solve the system

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

// Declare variables to eliminate the first variable
double Mp[16][2];
double Yp[16];
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;

}
```

5.2.2 3D static

Header

```
#ifndef __FMB3D_H_
#define __FMB3D_H_

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

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

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

```
// of the resulting AABB of FMBTestIntersection(B,A)
// 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
  *nbRemainRows = 0;

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

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

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

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

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

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

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

          if (iCol != iVar) {

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

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

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

        }

      }

      // Update the right side of the inequality
      Yp[*nbRemainRows] =
        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[*nbRemainRows] < sumNegCoeff - EPSILON) {

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

      }

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

    }

  }

}

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

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

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

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

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

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

      if (iCol != iVar) {

        MpnbRemainRows[jCol] = MiRow[iCol];
```

```
          ++jCol;

        }

      }

      Yp[*nbRemainRows] = Y[iRow];

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

    }

  }

  // If we reach here the system is not inconsistent
  return false;

}

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

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

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

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

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

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

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

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

        // Update the maximum bound
```

```
        *max = y;

      }

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

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

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

        // Update the minimum bound
        *min = y;

      }

    }

  }

}

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

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

  // Declare two variables to memorize the system to be solved M.X <= Y
  // (M arrangement is [iRow][iCol])
  double M[12][3];
  double Y[12];

  // Create the inequality system

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

  M[1][0] = -thoProj.comp[0][1];
  M[1][1] = -thoProj.comp[1][1];
```

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

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

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

M[4][0] = 0.0;
M[4][1] = -1.0;
M[4][2] = 0.0;
Y[4] = 0.0;

M[5][0] = 0.0;
M[5][1] = 0.0;
M[5][2] = -1.0;
Y[5] = 0.0;

// Variable to memorise the nb of rows in the system
int nbRows = 6;

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;

}

// Solve the system

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

// Declare variables to eliminate the first variable
double Mp[36][3];
double Yp[36];
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
double Mpp[324][3];
double Ypp[324];
int nbRowsPP;

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

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

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

}

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

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

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

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

```
  // Immediately return true
  return true;

}

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

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

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

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

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

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

```
    // Memorize the result
    *bdgBox = bdgBoxLocal;

  }

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

}
```

5.2.3 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
  *nbRemainRows = 0;

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

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

    // Shortcuts
    int sgnMIRowIVar = sgn(M[iRow][iVar]);
```

```
double fabsMIRowIVar = fabs(M[iRow][iVar]);
double YIRowDivideByFabsMIRowIVar = Y[iRow] / fabsMIRowIVar;

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

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

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

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

      if (iCol != iVar) {

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

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

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

      }

    }

    // Update the right side of the inequality
    Yp[*nbRemainRows] =
      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[*nbRemainRows] < sumNegCoeff - EPSILON) {

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

    }

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

  }
```

```
    }

  }

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

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

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

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

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

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

        if (iCol != iVar) {

          MpnbRemainRows[jCol] = MiRow[iCol];

          ++jCol;

        }

      }

      Yp[*nbRemainRows] = Y[iRow];

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

    }

  }

  // 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;

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

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

  // 0.0 <= t <= 1.0
  M[4][0] = 0.0;
  M[4][1] = 0.0;
  M[4][2] = 1.0;
  Y[4] = 1.0;

  M[5][0] = 0.0;
  M[5][1] = 0.0;
  M[5][2] = -1.0;
  Y[5] = 0.0;

  // Variable to memorise the nb of rows in the system
  int nbRows = 6;
```

```
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;

}

// Solve the system
```

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

// Declare variables to eliminate the first variable
double Mp[25][3];
double Yp[25];
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
double Mpp[169][3];
double Ypp[169];
int nbRowsPP;

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

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

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

}

// Get the bounds for the remaining third variable
GetBound2DTime(
  THD_VAR,
  Mpp,
  Ypp,
  nbRowsPP,
  &bdgBoxLocal);
```

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

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

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

  // Immediately return true
  return true;

}

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

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

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

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

```
      &nbRowsPP);

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

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

    // Memorize the result
    *bdgBox = bdgBoxLocal;

  }

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

}
```

5.2.4 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
*nbRemainRows = 0;

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

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

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

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

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

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

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

        if (iCol != iVar) {

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

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

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

        }

      }

      // Update the right side of the inequality
      Yp[*nbRemainRows] =
        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[*nbRemainRows] < sumNegCoeff - EPSILON) {

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

      }

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

    }

  }

}

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

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

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

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

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

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

      if (iCol != iVar) {

        MpnbRemainRows[jCol] = MiRow[iCol];

        ++jCol;

      }

    }

    Yp[*nbRemainRows] = Y[iRow];

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

  }

}

// If we reach here the system is not inconsistent
return false;
```

```
}

// Get the bounds of the 'iVar'-th variable in the 'nbRows' rows
// system 'M'.X<='Y' and store them in the 'iVar'-th axis of the
// AABB 'bdgBox'
// ('M' arrangement is [iRow][iCol])
// The system is supposed to have been reduced to only one variable
// per row, the one in argument
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBound3DTime(
     const int iVar,
  const double (*M)[4],
  const double* Y,
     const int nbRows,
   AABB3DTime* const bdgBox) {

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

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

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

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

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

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

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

        // Update the maximum bound
        *max = y;

      }

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

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

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

        // Update the minimum bound
        *min = y;
```

```
      }

    }

  }

}

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho,
        AABB3DTime* const bdgBox) {

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

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

  // Create the inequality system

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

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

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

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

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

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

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

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

} else {

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

}

if (tho->type == FrameCuboid) {

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

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

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

} else {

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

}

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

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

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

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

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

```
// Solve the system

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

// Declare variables to eliminate the first variable
double Mp[49][4];
double Yp[49];
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
double Mpp[625][4];
double Ypp[625];
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
double Mppp[97969][4];
double Yppp[97969];
int nbRowsPPP;
```

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

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

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

}

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

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

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

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

  // Immediately return true
  return true;

}

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

// Get the bounds for the remaining third variable
GetBound3DTime(
  THD_VAR,
  Mppp,
  Yppp,
```

```
  nbRowsPPP,
  &bdgBoxLocal);

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

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

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

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

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

GetBound3DTime(
  SND_VAR,
  Mppp,
```

```
    Yppp,
    nbRowsPPP,
    &bdgBoxLocal);

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

    // Memorize the result
    *bdgBox = bdgBoxLocal;

  }

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

}
```

6 Example of use

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

6.1 2D static

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

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

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

  // Create the two objects to be tested for intersection
  double origP2D[2] = {0.0, 0.0};
  double compP2D[2][2] = {
    {1.0, 0.0},  // First component
    {0.0, 1.0}}; // Second component
  Frame2D P2D =
    Frame2DCreateStatic(
      FrameCuboid,
      origP2D,
      compP2D);

  double origQ2D[2] = {0.0,0.0};
  double compQ2D[2][2] = {
    {1.0, 1.0},
    {-1.0, 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;

}
```

6.2 3D static

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

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

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

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

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

  // Declare a variable to memorize the result of the intersection
  // detection
  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;
}
```

6.3 2D dynamic

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

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

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

  // Create the two objects to be tested for intersection
  double origP2DTime[2] = {0.0, 0.0};
  double speedP2DTime[2] = {0.0, 0.0};
  double compP2DTime[2][2] = {
```

```
    {1.0, 0.0},  // First component
    {0.0, 1.0}}; // Second component
  Frame2DTime P2DTime =
    Frame2DTimeCreateStatic(
      FrameCuboid,
      origP2DTime,
      speedP2DTime,
      compP2DTime);

  double origQ2DTime[2] = {-1.0,0.0};
  double speedQ2DTime[2] = {1.0,0.0};
  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;
}
```

6.4 3D dynamic

```
// Include standard libraries
```

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

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

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

  // Create the two objects to be tested for intersection
  double origP3DTime[3] = {0.0, 0.0, 0.0};
  double speedP3DTime[3] = {0.0, 0.0, 0.0};
  double compP3DTime[3][3] = {
    {1.0, 0.0, 0.0},  // First component
    {0.0, 1.0, 0.0},  // Second component
    {0.0, 0.0, 1.0}}; // Third component
  Frame3DTime P3DTime =
    Frame3DTimeCreateStatic(
      FrameCuboid,
      origP3DTime,
      speedP3DTime,
      compP3DTime);

  double origQ3DTime[3] = {-1.0, 0.0, 0.0};
  double speedQ3DTime[3] = {1.0, 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;
}
```

7 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.203)

7.1 Code

7.1.1 2D static

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

// Include FMB and SAT algorithm library
#include "fmb2d.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

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

```
// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void Validation2D(
  const Param2D paramP,
  const Param2D paramQ) {

  // Create the two Frames
  Frame2D P =
    Frame2DCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.comp);

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

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

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

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

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

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

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

      // Stop the validation
      exit(0);
```

```
    }

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

      // Update the number of intersection
      nbInter++;

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

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

    }

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

  }

}

// Main function
void Validate2D(void) {

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

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

  // Initialize the number of intersection and no intersection
  nbInter = 0;
  nbNoInter = 0;

  // Loop on the tests
  for (unsigned long iTest = NB_TESTS;
       iTest--;) {

    // Create two random Frame definitions
    Param2D* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

      // 50% chance of being a Cuboid or a Tetrahedron
      if (rnd() < 0.5)
        param->type = FrameCuboid;
      else
        param->type = FrameTetrahedron;

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

        param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

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

```
          param->comp[iComp][iAxis] =
            -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

        }

      }

      param = &paramQ;

    }

    // Calculate the determinant of the Frames' components matrix
    double detP =
      paramP.comp[0][0] * paramP.comp[1][1] -
      paramP.comp[1][0] * paramP.comp[0][1];

    double detQ =
      paramQ.comp[0][0] * paramQ.comp[1][1] -
      paramQ.comp[1][0] * paramQ.comp[0][1];

    // If the determinants are not null, ie the Frame are not degenerate
    if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

      // Run the validation on the two Frames
      Validation2D(
        paramP,
        paramQ);

    }

  }

  // If we reached it means the validation was successfull
  // Print results
  printf("Validation2D has succeed.\n");
  printf("Tested %lu intersections ", nbInter);
  printf("and %lu no intersections\n", nbNoInter);

}

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

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

  return 0;
}
```

7.1.2 3D static

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

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

// Epsilon to detect degenerated triangles
```

```
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

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

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void Validation3D(
  const Param3D paramP,
  const Param3D paramQ) {

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

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

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

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

    // Test intersection with FMB
    bool isIntersectingFMB =
      FMBTestIntersection3D(
        that,
        tho,
        NULL);

    // Test intersection with SAT
    bool isIntersectingSAT =
      SATTestIntersection3D(
        that,
        tho);
```

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

      // Print the disagreement
      printf("Validation3D has failed\n");
      Frame3DPrint(that);
      printf(" against ");
      Frame3DPrint(tho);
      printf("\n");
      printf("FMB : ");
      if (isIntersectingFMB == false)
        printf("no ");
      printf("intersection\n");
      printf("SAT : ");
      if (isIntersectingSAT == false)
        printf("no ");
      printf("intersection\n");

      // Stop the validation
      exit(0);

    }

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

      // Update the number of intersection
      nbInter++;

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

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

    }

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

  }

}

void Validate3D(void) {

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

  // Declare two variables to memorize the arguments to the
  // Validation function
  Param3D paramP;
  Param3D paramQ;

  // Initialize the number of intersection and no intersection
  nbInter = 0;
  nbNoInter = 0;

  // Loop on the tests
  for (unsigned long iTest = NB_TESTS;
       iTest--;) {
```

```
  // Create two random Frame definitions
  Param3D* param = &paramP;
  for (int iParam = 2;
       iParam--;) {

    // 50% chance of being a Cuboid or a Tetrahedron
    if (rnd() < 0.5)
      param->type = FrameCuboid;
    else
      param->type = FrameTetrahedron;

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

      param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

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

        param->comp[iComp][iAxis] =
          -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      }

    }

    param = &paramQ;

  }

  // Calculate the determinant of the Frames' components matrix
  double detP =
    paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2]-
    paramP.comp[1][2] * paramP.comp[2][1]) -
    paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2]-
    paramP.comp[0][2] * paramP.comp[2][1]) +
    paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2]-
    paramP.comp[0][2] * paramP.comp[1][1]);

  double detQ =
    paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2]-
    paramQ.comp[1][2] * paramQ.comp[2][1]) -
    paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2]-
    paramQ.comp[0][2] * paramQ.comp[2][1]) +
    paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2]-
    paramQ.comp[0][2] * paramQ.comp[1][1]);

  // If the determinants are not null, ie the Frame are not degenerate
  if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Validation3D(
      paramP,
      paramQ);

  }

}

// If we reached it means the validation was successfull
// Print results
```

```
  printf("Validation3D has succeed.\n");
  printf("Tested %lu intersections ", nbInter);
  printf("and %lu no intersections\n", nbNoInter);

}

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

  printf("===== 3D static ======\n");
  Validate3D();

  return 0;
}
```

7.1.3 2D dynamic

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

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

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

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

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void Validation2DTime(
  const Param2DTime paramP,
  const Param2DTime paramQ) {

  // Create the two Frames
  Frame2DTime P =
    Frame2DTimeCreateStatic(
      paramP.type,
      paramP.orig,
```

```
    paramP.speed,
    paramP.comp);

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

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

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

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

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

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

    // Print the disagreement
    printf("Validation2D has failed\n");
    Frame2DTimePrint(that);
    printf(" against ");
    Frame2DTimePrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT == false)
      printf("no ");
    printf("intersection\n");

    // Stop the validation
    exit(0);

  }

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

    // Update the number of intersection
    nbInter++;

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

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

    }

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

  }

}

// Main function
void Validate2DTime(void) {

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

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

  // Initialize the number of intersection and no intersection
  nbInter = 0;
  nbNoInter = 0;

  // Loop on the tests
  for (unsigned long iTest = NB_TESTS;
       iTest--;) {

    // Create two random Frame definitions
    Param2DTime* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

      // 50% chance of being a Cuboid or a Tetrahedron
      if (rnd() < 0.5)
        param->type = FrameCuboid;
      else
        param->type = FrameTetrahedron;

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

        param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
        param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

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

          param->comp[iComp][iAxis] =
            -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

        }

      }

      param = &paramQ;
```

```
    }

    // Calculate the determinant of the Frames' components matrix
    double detP =
      paramP.comp[0][0] * paramP.comp[1][1] -
      paramP.comp[1][0] * paramP.comp[0][1];

    double detQ =
      paramQ.comp[0][0] * paramQ.comp[1][1] -
      paramQ.comp[1][0] * paramQ.comp[0][1];

    // If the determinants are not null, ie the Frame are not degenerate
    if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

      // Run the validation on the two Frames
      Validation2DTime(
        paramP,
        paramQ);

    }

  }

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

}

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

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

  return 0;
}
```

7.1.4 3D dynamic

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

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

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)
```

```
// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

// Helper structure to pass arguments to the Validation function
typedef struct {
  FrameType type;
  double orig[3];
  double comp[3][3];
  double speed[3];
} Param3DTime;

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void Validation3DTime(
  const Param3DTime paramP,
  const Param3DTime paramQ) {

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

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

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

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

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

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

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

      // Print the disagreement
      printf("Validation3D has failed\n");
      Frame3DTimePrint(that);
```

```
      printf(" against ");
      Frame3DTimePrint(tho);
      printf("\n");
      printf("FMB : ");
      if (isIntersectingFMB == false)
        printf("no ");
      printf("intersection\n");
      printf("SAT : ");
      if (isIntersectingSAT == false)
        printf("no ");
      printf("intersection\n");

      // Stop the validation
      exit(0);

    }

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

      // Update the number of intersection
      nbInter++;

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

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

    }

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

  }

}

// Main function
void Validate3DTime(void) {

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

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

  // Initialize the number of intersection and no intersection
  nbInter = 0;
  nbNoInter = 0;

  // Loop on the tests
  for (unsigned long iTest = NB_TESTS;
       iTest--;) {

    // Create two random Frame definitions
    Param3DTime* param = &paramP;
    for (int iParam = 2;
         iParam--;) {
```

```
    // 50% chance of being a Cuboid or a Tetrahedron
    if (rnd() < 0.5)
      param->type = FrameCuboid;
    else
      param->type = FrameTetrahedron;

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

      param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
      param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

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

        param->comp[iComp][iAxis] =
          -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      }

    }

    param = &paramQ;

  }

  // Calculate the determinant of the Frames' components matrix
  double detP =
    paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2]-
    paramP.comp[1][2] * paramP.comp[2][1]) -
    paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2]-
    paramP.comp[0][2] * paramP.comp[2][1]) +
    paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2]-
    paramP.comp[0][2] * paramP.comp[1][1]);

  double detQ =
    paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2]-
    paramQ.comp[1][2] * paramQ.comp[2][1]) -
    paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2]-
    paramQ.comp[0][2] * paramQ.comp[2][1]) +
    paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2]-
    paramQ.comp[0][2] * paramQ.comp[1][1]);

  // If the determinants are not null, ie the Frame are not degenerate
  if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Validation3DTime(
      paramP,
      paramQ);

  }

}

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

```
}

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

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

  return 0;
}
```

7.2 Results

7.2.1 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(1.000000,1.000000) x(-1.000000,0.000000) y(0.000000,-1.000000)
 Failed
Expected : no intersection
Got : intersection
```

7.2.2 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(-1.000000,-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(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
    (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
 Failed
Expected : no intersection
Got : intersection
```

7.2.3 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(-1.000000,-1.000000) s(1.000000,0.000000) x(1.000000,0.000000) y
    (0.000000,1.000000)
 Failed
Expected : no intersection
Got : intersection
```

7.2.4 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(-1.000000,-1.000000,0.000000) 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)
 Failed
Expected : no intersection
Got : intersection
```

8 Validation

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.203)

8.1 Code

8.1.1 2D static

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

// Include FMB and SAT algorithm library
#include "fmb2d.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

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

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void Validation2D(
  const Param2D paramP,
  const Param2D paramQ) {

  // Create the two Frames
  Frame2D P =
    Frame2DCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.comp);

  Frame2D Q =
    Frame2DCreateStatic(
      paramQ.type,
      paramQ.orig,
```

```
    paramQ.comp);

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

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

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

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

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

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

    // Stop the validation
    exit(0);

  }

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

    // Update the number of intersection
    nbInter++;

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

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

  }

  // Flip the pair of Frames
  that = &Q;
```

```
    tho = &P;

  }

}

// Main function
void Validate2D(void) {

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

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

  // Initialize the number of intersection and no intersection
  nbInter = 0;
  nbNoInter = 0;

  // Loop on the tests
  for (unsigned long iTest = NB_TESTS;
       iTest--;) {

    // Create two random Frame definitions
    Param2D* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

      // 50% chance of being a Cuboid or a Tetrahedron
      if (rnd() < 0.5)
        param->type = FrameCuboid;
      else
        param->type = FrameTetrahedron;

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

        param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

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

          param->comp[iComp][iAxis] =
            -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

        }

      }

      param = &paramQ;

    }

    // Calculate the determinant of the Frames' components matrix
    double detP =
      paramP.comp[0][0] * paramP.comp[1][1] -
      paramP.comp[1][0] * paramP.comp[0][1];

    double detQ =
      paramQ.comp[0][0] * paramQ.comp[1][1] -
```

```
      paramQ.comp[1][0] * paramQ.comp[0][1];

    // If the determinants are not null, ie the Frame are not degenerate
    if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

      // Run the validation on the two Frames
      Validation2D(
        paramP,
        paramQ);

    }

  }

  // If we reached it means the validation was successfull
  // Print results
  printf("Validation2D has succeed.\n");
  printf("Tested %lu intersections ", nbInter);
  printf("and %lu no intersections\n", nbNoInter);

}

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

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

  return 0;
}
```

8.1.2 3D static

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

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

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

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

```
  double comp[3][3];
} Param3D;

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void Validation3D(
  const Param3D paramP,
  const Param3D paramQ) {

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

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

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

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

    // Test intersection with FMB
    bool isIntersectingFMB =
      FMBTestIntersection3D(
        that,
        tho,
        NULL);

    // Test intersection with SAT
    bool isIntersectingSAT =
      SATTestIntersection3D(
        that,
        tho);

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

      // Print the disagreement
      printf("Validation3D has failed\n");
      Frame3DPrint(that);
      printf(" against ");
      Frame3DPrint(tho);
      printf("\n");
      printf("FMB : ");
      if (isIntersectingFMB == false)
        printf("no ");
      printf("intersection\n");
      printf("SAT : ");
      if (isIntersectingSAT == false)
        printf("no ");
      printf("intersection\n");
```

```
      // Stop the validation
      exit(0);

    }

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

      // Update the number of intersection
      nbInter++;

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

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

    }

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

  }

}

void Validate3D(void) {

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

  // Declare two variables to memorize the arguments to the
  // Validation function
  Param3D paramP;
  Param3D paramQ;

  // Initialize the number of intersection and no intersection
  nbInter = 0;
  nbNoInter = 0;

  // Loop on the tests
  for (unsigned long iTest = NB_TESTS;
       iTest--;) {

    // Create two random Frame definitions
    Param3D* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

      // 50% chance of being a Cuboid or a Tetrahedron
      if (rnd() < 0.5)
        param->type = FrameCuboid;
      else
        param->type = FrameTetrahedron;

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

        param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

        for (int iComp = 3;
```

```
          iComp--;) {

        param->comp[iComp][iAxis] =
          -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      }

    }

    param = &paramQ;

  }

  // Calculate the determinant of the Frames' components matrix
  double detP =
    paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2]-
    paramP.comp[1][2] * paramP.comp[2][1]) -
    paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2]-
    paramP.comp[0][2] * paramP.comp[2][1]) +
    paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2]-
    paramP.comp[0][2] * paramP.comp[1][1]);

  double detQ =
    paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2]-
    paramQ.comp[1][2] * paramQ.comp[2][1]) -
    paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2]-
    paramQ.comp[0][2] * paramQ.comp[2][1]) +
    paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2]-
    paramQ.comp[0][2] * paramQ.comp[1][1]);

  // If the determinants are not null, ie the Frame are not degenerate
  if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Validation3D(
      paramP,
      paramQ);

  }

}

// If we reached it means the validation was successfull
// Print results
printf("Validation3D has succeed.\n");
printf("Tested %lu intersections ", nbInter);
printf("and %lu no intersections\n", nbNoInter);

}

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

  printf("===== 3D static ======\n");
  Validate3D();

  return 0;
}
```

8.1.3 2D dynamic

```
// Include standard libraries
```

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

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

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

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

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void Validation2DTime(
  const Param2DTime paramP,
  const Param2DTime paramQ) {

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

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

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

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

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

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

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

      // Print the disagreement
      printf("Validation2D has failed\n");
      Frame2DTimePrint(that);
      printf(" against ");
      Frame2DTimePrint(tho);
      printf("\n");
      printf("FMB : ");
      if (isIntersectingFMB == false)
        printf("no ");
      printf("intersection\n");
      printf("SAT : ");
      if (isIntersectingSAT == false)
        printf("no ");
      printf("intersection\n");

      // Stop the validation
      exit(0);

    }

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

      // Update the number of intersection
      nbInter++;

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

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

    }

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

  }

}

// Main function
void Validate2DTime(void) {

  // Initialise the random generator
```

```
srandom(time(NULL));

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

// Initialize the number of intersection and no intersection
nbInter = 0;
nbNoInter = 0;

// Loop on the tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

  // Create two random Frame definitions
  Param2DTime* param = &paramP;
  for (int iParam = 2;
       iParam--;) {

    // 50% chance of being a Cuboid or a Tetrahedron
    if (rnd() < 0.5)
      param->type = FrameCuboid;
    else
      param->type = FrameTetrahedron;

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

      param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
      param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

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

        param->comp[iComp][iAxis] =
          -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      }

    }

    param = &paramQ;

  }

  // Calculate the determinant of the Frames' components matrix
  double detP =
    paramP.comp[0][0] * paramP.comp[1][1] -
    paramP.comp[1][0] * paramP.comp[0][1];

  double detQ =
    paramQ.comp[0][0] * paramQ.comp[1][1] -
    paramQ.comp[1][0] * paramQ.comp[0][1];

  // If the determinants are not null, ie the Frame are not degenerate
  if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Validation2DTime(
      paramP,
      paramQ);
```

```
    }

  }

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

}

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

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

  return 0;
}
```

8.1.4 3D dynamic

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

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

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

// Helper structure to pass arguments to the Validation function
typedef struct {
  FrameType type;
  double orig[3];
  double comp[3][3];
  double speed[3];
} Param3DTime;

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void Validation3DTime(
  const Param3DTime paramP,
```

```
const Param3DTime paramQ) {

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

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

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

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

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

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

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

    // Print the disagreement
    printf("Validation3D has failed\n");
    Frame3DTimePrint(that);
    printf(" against ");
    Frame3DTimePrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT == false)
      printf("no ");
    printf("intersection\n");

    // Stop the validation
    exit(0);

  }

  // If the Frames are in intersection
```

```
    if (isIntersectingFMB == true) {

      // Update the number of intersection
      nbInter++;

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

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

    }

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

  }

}

// Main function
void Validate3DTime(void) {

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

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

  // Initialize the number of intersection and no intersection
  nbInter = 0;
  nbNoInter = 0;

  // Loop on the tests
  for (unsigned long iTest = NB_TESTS;
       iTest--;) {

    // Create two random Frame definitions
    Param3DTime* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

      // 50% chance of being a Cuboid or a Tetrahedron
      if (rnd() < 0.5)
        param->type = FrameCuboid;
      else
        param->type = FrameTetrahedron;

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

        param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
        param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

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

          param->comp[iComp][iAxis] =
            -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
```

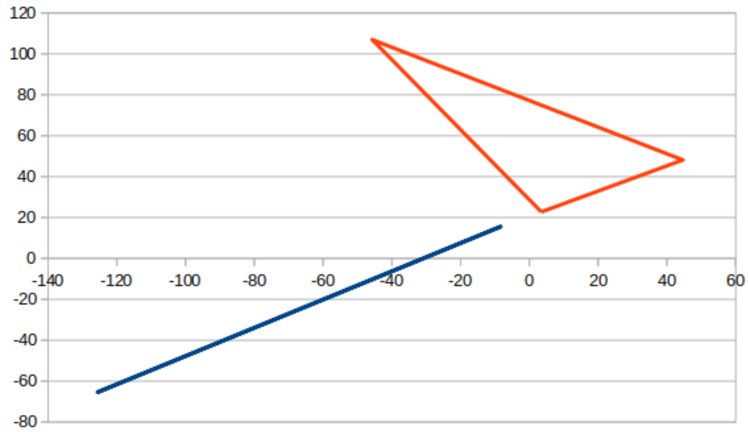
```
        }

      }

      param = &paramQ;

    }

    // Calculate the determinant of the Frames' components matrix
    double detP =
      paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2]-
      paramP.comp[1][2] * paramP.comp[2][1]) -
      paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2]-
      paramP.comp[0][2] * paramP.comp[2][1]) +
      paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2]-
      paramP.comp[0][2] * paramP.comp[1][1]);

    double detQ =
      paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2]-
      paramQ.comp[1][2] * paramQ.comp[2][1]) -
      paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2]-
      paramQ.comp[0][2] * paramQ.comp[2][1]) +
      paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2]-
      paramQ.comp[0][2] * paramQ.comp[1][1]);

    // If the determinants are not null, ie the Frame are not degenerate
    if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

      // Run the validation on the two Frames
      Validation3DTime(
        paramP,
        paramQ);

    }

  }

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

}

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

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

  return 0;
}
```

8.2 Results

8.2.1 Failures

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

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



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

8.2.2 2D static

```
===== 2D static ======
Validation2D has succeed.
Tested 467954 intersections and 1531962 no intersections
```

8.2.3 2D dynamic

```
===== 2D dynamic ======
Validation2DTime has succeed.
Tested 741478 intersections and 1258454 no intersections
```

8.2.4 3D static

```
===== 3D static ======
Validation3D has succeed.
Tested 315464 intersections and 1684532 no intersections
```

8.2.5 3D dynamic

```
===== 3D dynamic ======
Validation3DTime has succeed.
Tested 523506 intersections and 1476492 no intersections
```

9 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.

9.1 Code

9.1.1 2D static

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

// Include FMB and SAT algorithm library
#include "fmb2d.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 100000
// Nb of times the test is run on one pair of frame, used to
// slow down the processus and be able to measure time
#define NB_REPEAT_2D 1500

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

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

```
// Global variables to count nb of tests resulting in intersection
// and no intersection, and min/max/total time of execution for each
double minInter;
double maxInter;
double sumInter;
unsigned long countInter;
double minNoInter;
double maxNoInter;
double sumNoInter;
unsigned long countNoInter;

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

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

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

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

// Qualification function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and measure the time of execution of each
void Qualification2DStatic(
        const Param2D paramP,
        const Param2D paramQ) {

  // Create the two Frames
  Frame2D P =
    Frame2DCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.comp);

  Frame2D Q =
```

```
  Frame2DCreateStatic(
    paramQ.type,
    paramQ.orig,
    paramQ.comp);

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

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

  // Declare an array to memorize the results of the repeated
  // test on the same pair,
  // to prevent optimization from the compiler to remove the for loop
  bool isIntersectingFMB[NB_REPEAT_2D] = {false};

  // Start measuring time
  struct timeval start;
  gettimeofday(&start, NULL);

  // Run the FMB intersection test
  for (int i = NB_REPEAT_2D;
       i--;) {

    isIntersectingFMB[i] =
      FMBTestIntersection2D(
        that,
        tho,
        NULL);
  }

  // Stop measuring time
  struct timeval stop;
  gettimeofday(&stop, NULL);

  // Calculate the delay of execution
  unsigned long deltausFMB = 0;
  if (stop.tv_sec < start.tv_sec) {
    printf("time warps, try again\n");
    exit(0);
  }
  if (stop.tv_sec > start.tv_sec + 1) {
    printf("deltausFMB >> 1s, decrease NB_REPEAT\n");
    exit(0);
  }
  if (stop.tv_usec < start.tv_usec) {
    deltausFMB = stop.tv_sec - start.tv_sec;
    deltausFMB += stop.tv_usec + 1000000 - start.tv_usec;
  } else {
    deltausFMB = stop.tv_usec - start.tv_usec;
  }

  // Declare an array to memorize the results of the repeated
  // test on the same pair,
  // to prevent optimization from the compiler to remove the for loop
  bool isIntersectingSAT[NB_REPEAT_2D] = {false};

  // Start measuring time
  gettimeofday(&start, NULL);
```

```
// Run the FMB intersection test
for (int i = NB_REPEAT_2D;
     i--;) {

  isIntersectingSAT[i] =
    SATTestIntersection2D(
      that,
      tho);

}

// Stop measuring time
gettimeofday(&stop, NULL);

// Calculate the delay of execution
unsigned long deltausSAT = 0;
if (stop.tv_sec < start.tv_sec) {
  printf("time warps, try again\n");
  exit(0);
}
if (stop.tv_sec > start.tv_sec + 1) {
  printf("deltausSAT >> 1s, decrease NB_REPEAT\n");
  exit(0);
}
if (stop.tv_usec < start.tv_usec) {
  deltausSAT = stop.tv_sec - start.tv_sec;
  deltausSAT += stop.tv_usec + 1000000 - start.tv_usec;
} else {
  deltausSAT = stop.tv_usec - start.tv_usec;
}

// If the delays are greater than 10ms
if (deltausFMB >= 10 && deltausSAT >= 10) {

  // If FMB and SAT disagrees
  if (isIntersectingFMB[0] != isIntersectingSAT[0]) {

    printf("Qualification has failed\n");
    Frame2DPrint(that);
    printf(" against ");
    Frame2DPrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB[0] == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT[0] == false)
      printf("no ");
    printf("intersection\n");

    // Stop the qualification test
    exit(0);

  }

  // Get the ratio of execution time
  double ratio = ((double)deltausFMB) / ((double)deltausSAT);

  // If the Frames intersect
  if (isIntersectingSAT[0] == true) {
```

```
// Update the counters
if (countInter == 0) {

  minInter = ratio;
  maxInter = ratio;

} else {

  if (minInter > ratio)
    minInter = ratio;
  if (maxInter < ratio)
    maxInter = ratio;

}
sumInter += ratio;
++countInter;

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

  if (countInterCC == 0) {

    minInterCC = ratio;
    maxInterCC = ratio;

  } else {

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

  }
  sumInterCC += ratio;
  ++countInterCC;

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

  if (countInterCT == 0) {

    minInterCT = ratio;
    maxInterCT = ratio;

  } else {

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

  }
  sumInterCT += ratio;
  ++countInterCT;

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

  if (countInterTC == 0) {

    minInterTC = ratio;
    maxInterTC = ratio;
```

```
    } else {

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

    }
    sumInterTC += ratio;
    ++countInterTC;

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

    if (countInterTT == 0) {

      minInterTT = ratio;
      maxInterTT = ratio;

    } else {

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

    }
    sumInterTT += ratio;
    ++countInterTT;

  }

// Else, the Frames do not intersect
} else {

  // Update the counters
  if (countNoInter == 0) {

    minNoInter = ratio;
    maxNoInter = ratio;

  } else {

    if (minNoInter > ratio)
      minNoInter = ratio;
    if (maxNoInter < ratio)
      maxNoInter = ratio;

  }
  sumNoInter += ratio;
  ++countNoInter;

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

    if (countNoInterCC == 0) {

      minNoInterCC = ratio;
      maxNoInterCC = ratio;

    } else {
```

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

  }
  sumNoInterCC += ratio;
  ++countNoInterCC;

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

  if (countNoInterCT == 0) {

    minNoInterCT = ratio;
    maxNoInterCT = ratio;

  } else {

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

  }
  sumNoInterCT += ratio;
  ++countNoInterCT;

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

  if (countNoInterTC == 0) {

    minNoInterTC = ratio;
    maxNoInterTC = ratio;

  } else {

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

  }
  sumNoInterTC += ratio;
  ++countNoInterTC;

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

  if (countNoInterTT == 0) {

    minNoInterTT = ratio;
    maxNoInterTT = ratio;

  } else {

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

```
          }
          sumNoInterTT += ratio;
          ++countNoInterTT;

        }
      }

    // Else, if time of execution for FMB was less than a 10ms
    } else if (deltausFMB < 10) {

      printf("deltausFMB < 10ms, increase NB_REPEAT\n");
      exit(0);

    // Else, if time of execution for SAT was less than a 10ms
    } else if (deltausSAT < 10) {

      printf("deltausSAT < 10ms, increase NB_REPEAT\n");
      exit(0);

    }

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

  }

}

void Qualify2DStatic(void) {

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

  // Loop on runs
  for (int iRun = 0;
       iRun < NB_RUNS;
       ++iRun) {

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

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

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

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

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

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

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

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

  // Create two random Frame definitions
  Param2D* param = &paramP;
  for (int iParam = 2;
       iParam--;) {

    // 50% chance of being a Cuboid or a Tetrahedron
    if (rnd() < 0.5)
      param->type = FrameCuboid;
    else
      param->type = FrameTetrahedron;

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

      param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

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

        param->comp[iComp][iAxis] =
          -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      }

    }
```

```
    param = &paramQ;

  }

  // Calculate the determinant of the Frames' components matrix

  double detP =
    paramP.comp[0][0] * paramP.comp[1][1] -
    paramP.comp[1][0] * paramP.comp[0][1];

  double detQ =
    paramQ.comp[0][0] * paramQ.comp[1][1] -
    paramQ.comp[1][0] * paramQ.comp[0][1];

  // If the determinants are not null, ie the Frame are not degenerate
  if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Qualification2DStatic(
      paramP,
      paramQ);

  }

}

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

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

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

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

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

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

}

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

printf("%lu\t%lu\t", countInter, countNoInter);
double avgInter = sumInter / (double)countInter;
printf("%f\t%f\t%f\t", minInter, avgInter, maxInter);
```

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

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

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

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

    printf("%lu\t%lu\t", countInterTT, countNoInterTT);
    double avgInterTT = sumInterTT / (double)countInterTT;
    printf("%f\t%f\t%f\t", minInterTT, avgInterTT, maxInterTT);
    double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
    printf("%f\t%f\t%f\t", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
      ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    printf("%f\t%f\t%f\n",
      (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
      avgTT,
      (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));

  }

}

int main(int argc, char** argv) {
```

```
  Qualify2DStatic();

  return 0;
}
```

9.1.2 3D static

```
// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>

// Include FMB and SAT algorithm library
#include "fmb3d.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 100000
// Nb of times the test is run on one pair of frame, used to
// slow down the processus and be able to measure time
#define NB_REPEAT_3D 800

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Helper structure to pass arguments to the Qualification function
typedef struct {
  FrameType type;
  double orig[3];
  double comp[3][3];
} Param3D;

// Global variables to count nb of tests resulting in intersection
// and no intersection, and min/max/total time of execution for each
double minInter;
double maxInter;
double sumInter;
unsigned long countInter;
double minNoInter;
double maxNoInter;
double sumNoInter;
unsigned long countNoInter;

double minInterCC;
double maxInterCC;
double sumInterCC;
unsigned long countInterCC;
double minNoInterCC;
double maxNoInterCC;
double sumNoInterCC;
unsigned long countNoInterCC;
```

```
double minInterCT;
double maxInterCT;
double sumInterCT;
unsigned long countInterCT;
double minNoInterCT;
double maxNoInterCT;
double sumNoInterCT;
unsigned long countNoInterCT;

double minInterTC;
double maxInterTC;
double sumInterTC;
unsigned long countInterTC;
double minNoInterTC;
double maxNoInterTC;
double sumNoInterTC;
unsigned long countNoInterTC;

double minInterTT;
double maxInterTT;
double sumInterTT;
unsigned long countInterTT;
double minNoInterTT;
double maxNoInterTT;
double sumNoInterTT;
unsigned long countNoInterTT;

// Qualification function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and measure the time of execution of each
void Qualification3DStatic(
        const Param3D paramP,
        const Param3D paramQ) {

  // Create the two Frames
  Frame3D P =
    Frame3DCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.comp);

  Frame3D Q =
    Frame3DCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.comp);

  // Helper variables to loop on the pair (that, tho) and (tho, that)
  Frame3D* that = &P;
  Frame3D* tho = &Q;

  // Loop on pairs of Frames
  for (int iPair = 2;
       iPair--;) {

    // Declare an array to memorize the results of the repeated
    // test on the same pair,
    // to prevent optimization from the compiler to remove the for loop
    bool isIntersectingFMB[NB_REPEAT_3D] = {false};

    // Start measuring time
    struct timeval start;
```

```
gettimeofday(&start, NULL);

// Run the FMB intersection test
for (int i = NB_REPEAT_3D;
     i--;) {

  isIntersectingFMB[i] =
    FMBTestIntersection3D(
      that,
      tho,
      NULL);
}

// Stop measuring time
struct timeval stop;
gettimeofday(&stop, NULL);

// Calculate the delay of execution
unsigned long deltausFMB = 0;
if (stop.tv_sec < start.tv_sec) {
  printf("time warps, try again\n");
  exit(0);
}
if (stop.tv_sec > start.tv_sec + 1) {
  printf("deltausFMB >> 1s, decrease NB_REPEAT\n");
  exit(0);
}
if (stop.tv_usec < start.tv_usec) {
  deltausFMB = stop.tv_sec - start.tv_sec;
  deltausFMB += stop.tv_usec + 1000000 - start.tv_usec;
} else {
  deltausFMB = stop.tv_usec - start.tv_usec;
}

// Declare an array to memorize the results of the repeated
// test on the same pair,
// to prevent optimization from the compiler to remove the for loop
bool isIntersectingSAT[NB_REPEAT_3D] = {false};

// Start measuring time
gettimeofday(&start, NULL);

// Run the FMB intersection test
for (int i = NB_REPEAT_3D;
     i--;) {

  isIntersectingSAT[i] =
    SATTestIntersection3D(
      that,
      tho);

}

// Stop measuring time
gettimeofday(&stop, NULL);

// Calculate the delay of execution
unsigned long deltausSAT = 0;
if (stop.tv_sec < start.tv_sec) {
  printf("time warps, try again\n");
  exit(0);
}
```

```
if (stop.tv_sec > start.tv_sec + 1) {
  printf("deltausSAT >> 1s, decrease NB_REPEAT\n");
  exit(0);
}
if (stop.tv_usec < start.tv_usec) {
  deltausSAT = stop.tv_sec - start.tv_sec;
  deltausSAT += stop.tv_usec + 1000000 - start.tv_usec;
} else {
  deltausSAT = stop.tv_usec - start.tv_usec;
}

// If the delays are greater than 10ms
if (deltausFMB >= 10 && deltausSAT >= 10) {

  // If FMB and SAT disagrees
  if (isIntersectingFMB[0] != isIntersectingSAT[0]) {

    printf("Qualification has failed\n");
    Frame3DPrint(that);
    printf(" against ");
    Frame3DPrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB[0] == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT[0] == false)
      printf("no ");
    printf("intersection\n");

    // Stop the qualification test
    exit(0);

  }

  // Get the ratio of execution time
  double ratio = ((double)deltausFMB) / ((double)deltausSAT);

  // If the Frames intersect
  if (isIntersectingSAT[0] == true) {

    // Update the counters
    if (countInter == 0) {

      minInter = ratio;
      maxInter = ratio;

    } else {

      if (minInter > ratio)
        minInter = ratio;
      if (maxInter < ratio)
        maxInter = ratio;

    }
    sumInter += ratio;
    ++countInter;

    if (paramP.type == FrameCuboid &&
        paramQ.type == FrameCuboid) {
```

```
  if (countInterCC == 0) {

    minInterCC = ratio;
    maxInterCC = ratio;

  } else {

    if (minInterCC > ratio)
      minInterCC = ratio;
    if (maxInterCC < ratio)
      maxInterCC = ratio;

  }
  sumInterCC += ratio;
  ++countInterCC;

} else if (paramP.type == FrameCuboid &&
           paramQ.type == FrameTetrahedron) {

  if (countInterCT == 0) {

    minInterCT = ratio;
    maxInterCT = ratio;

  } else {

    if (minInterCT > ratio)
      minInterCT = ratio;
    if (maxInterCT < ratio)
      maxInterCT = ratio;

  }
  sumInterCT += ratio;
  ++countInterCT;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameCuboid) {

  if (countInterTC == 0) {

    minInterTC = ratio;
    maxInterTC = ratio;

  } else {

    if (minInterTC > ratio)
      minInterTC = ratio;
    if (maxInterTC < ratio)
      maxInterTC = ratio;

  }
  sumInterTC += ratio;
  ++countInterTC;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameTetrahedron) {

  if (countInterTT == 0) {

    minInterTT = ratio;
    maxInterTT = ratio;
```

```
    } else {

      if (minInterTT > ratio)
        minInterTT = ratio;
      if (maxInterTT < ratio)
        maxInterTT = ratio;

    }
    sumInterTT += ratio;
    ++countInterTT;

  }

// Else, the Frames do not intersect
} else {

  // Update the counters
  if (countNoInter == 0) {

    minNoInter = ratio;
    maxNoInter = ratio;

  } else {

    if (minNoInter > ratio)
      minNoInter = ratio;
    if (maxNoInter < ratio)
      maxNoInter = ratio;

  }
  sumNoInter += ratio;
  ++countNoInter;

  if (paramP.type == FrameCuboid &&
      paramQ.type == FrameCuboid) {

    if (countNoInterCC == 0) {

      minNoInterCC = ratio;
      maxNoInterCC = ratio;

    } else {

      if (minNoInterCC > ratio)
        minNoInterCC = ratio;
      if (maxNoInterCC < ratio)
        maxNoInterCC = ratio;

    }
    sumNoInterCC += ratio;
    ++countNoInterCC;

  } else if (paramP.type == FrameCuboid &&
             paramQ.type == FrameTetrahedron) {

    if (countNoInterCT == 0) {

      minNoInterCT = ratio;
      maxNoInterCT = ratio;

    } else {
```

```
        if (minNoInterCT > ratio)
          minNoInterCT = ratio;
        if (maxNoInterCT < ratio)
          maxNoInterCT = ratio;

      }
      sumNoInterCT += ratio;
      ++countNoInterCT;

    } else if (paramP.type == FrameTetrahedron &&
               paramQ.type == FrameCuboid) {

      if (countNoInterTC == 0) {

        minNoInterTC = ratio;
        maxNoInterTC = ratio;

      } else {

        if (minNoInterTC > ratio)
          minNoInterTC = ratio;
        if (maxNoInterTC < ratio)
          maxNoInterTC = ratio;

      }
      sumNoInterTC += ratio;
      ++countNoInterTC;

    } else if (paramP.type == FrameTetrahedron &&
               paramQ.type == FrameTetrahedron) {

      if (countNoInterTT == 0) {

        minNoInterTT = ratio;
        maxNoInterTT = ratio;

      } else {

        if (minNoInterTT > ratio)
          minNoInterTT = ratio;
        if (maxNoInterTT < ratio)
          maxNoInterTT = ratio;

      }
      sumNoInterTT += ratio;
      ++countNoInterTT;

    }
  }

// Else, if time of execution for FMB was less than a 10ms
} else if (deltausFMB < 10) {

  printf("deltausFMB < 10ms, increase NB_REPEAT\n");
  exit(0);

// Else, if time of execution for SAT was less than a 10ms
} else if (deltausSAT < 10) {

  printf("deltausSAT < 10ms, increase NB_REPEAT\n");
  exit(0);
```

```
    }

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

  }

}

void Qualify3DStatic(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Loop on runs
  for (int iRun = 0;
       iRun < NB_RUNS;
       ++iRun) {

    // Ratio intersection/no intersection for the displayed results
    double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

    // Initialize counters
    minInter = 0.0;
    maxInter = 0.0;
    sumInter = 0.0;
    countInter = 0;
    minNoInter = 0.0;
    maxNoInter = 0.0;
    sumNoInter = 0.0;
    countNoInter = 0;

    minInterCC = 0.0;
    maxInterCC = 0.0;
    sumInterCC = 0.0;
    countInterCC = 0;
    minNoInterCC = 0.0;
    maxNoInterCC = 0.0;
    sumNoInterCC = 0.0;
    countNoInterCC = 0;

    minInterCT = 0.0;
    maxInterCT = 0.0;
    sumInterCT = 0.0;
    countInterCT = 0;
    minNoInterCT = 0.0;
    maxNoInterCT = 0.0;
    sumNoInterCT = 0.0;
    countNoInterCT = 0;

    minInterTC = 0.0;
    maxInterTC = 0.0;
    sumInterTC = 0.0;
    countInterTC = 0;
    minNoInterTC = 0.0;
    maxNoInterTC = 0.0;
    sumNoInterTC = 0.0;
    countNoInterTC = 0;

    minInterTT = 0.0;
    maxInterTT = 0.0;
```

```
sumInterTT = 0.0;
countInterTT = 0;
minNoInterTT = 0.0;
maxNoInterTT = 0.0;
sumNoInterTT = 0.0;
countNoInterTT = 0;

// Declare two variables to memozie the arguments to the
// Qualification function
Param3D paramP;
Param3D paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

  // Create two random Frame definitions
  Param3D* param = &paramP;
  for (int iParam = 2;
       iParam--;) {

    // 50% chance of being a Cuboid or a Tetrahedron
    if (rnd() < 0.5)
      param->type = FrameCuboid;
    else
      param->type = FrameTetrahedron;

    for (int iAxis = 3;
         iAxis--;) {

      param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      for (int iComp = 3;
           iComp--;) {

        param->comp[iComp][iAxis] =
          -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      }

    }

    param = &paramQ;

  }

  // Calculate the determinant of the Frames' components matrix
  double detP =
    paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2]-
    paramP.comp[1][2] * paramP.comp[2][1]) -
    paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2]-
    paramP.comp[0][2] * paramP.comp[2][1]) +
    paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2]-
    paramP.comp[0][2] * paramP.comp[1][1]);

  double detQ =
    paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2]-
    paramQ.comp[1][2] * paramQ.comp[2][1]) -
    paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2]-
    paramQ.comp[0][2] * paramQ.comp[2][1]) +
    paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2]-
    paramQ.comp[0][2] * paramQ.comp[1][1]);
```

```
  // If the determinants are not null, ie the Frame are not degenerate
  if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Qualification3DStatic(
      paramP,
      paramQ);

  }

}

// Display the results
if (iRun == 0) {

  printf("percPairInter\t");
  printf("countInter\tcountNoInter\t");
  printf("minInter\tavgInter\tmaxInter\t");
  printf("minNoInter\tavgNoInter\tmaxNoInter\t");
  printf("minTotal\tavgTotal\tmaxTotal\t");

  printf("countInterCC\tcountNoInterCC\t");
  printf("minInterCC\tavgInterCC\tmaxInterCC\t");
  printf("minNoInterCC\tavgNoInterCC\tmaxNoInterCC\t");
  printf("minTotalCC\tavgTotalCC\tmaxTotalCC\t");

  printf("countInterCT\tcountNoInterCT\t");
  printf("minInterCT\tavgInterCT\tmaxInterCT\t");
  printf("minNoInterCT\tavgNoInterCT\tmaxNoInterCT\t");
  printf("minTotalCT\tavgTotalCT\tmaxTotalCT\t");

  printf("countInterTC\tcountNoInterTC\t");
  printf("minInterTC\tavgInterTC\tmaxInterTC\t");
  printf("minNoInterTC\tavgNoInterTC\tmaxNoInterTC\t");
  printf("minTotalTC\tavgTotalTC\tmaxTotalTC\t");

  printf("countInterTT\tcountNoInterTT\t");
  printf("minInterTT\tavgInterTT\tmaxInterTT\t");
  printf("minNoInterTT\tavgNoInterTT\tmaxNoInterTT\t");
  printf("minTotalTT\tavgTotalTT\tmaxTotalTT\n");

}

printf("%.1f\t", ratioInter);

printf("%lu\t%lu\t", countInter, countNoInter);
double avgInter = sumInter / (double)countInter;
printf("%f\t%f\t%f\t", minInter, avgInter, maxInter);
double avgNoInter = sumNoInter / (double)countNoInter;
printf("%f\t%f\t%f\t", minNoInter, avgNoInter, maxNoInter);
double avg =
  ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
printf("%f\t%f\t%f\t",
  (minNoInter < minInter ? minNoInter : minInter),
  avg,
  (maxNoInter > maxInter ? maxNoInter : maxInter));

printf("%lu\t%lu\t", countInterCC, countNoInterCC);
double avgInterCC = sumInterCC / (double)countInterCC;
printf("%f\t%f\t%f\t", minInterCC, avgInterCC, maxInterCC);
double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
```

```
    printf("%f\t%f\t%f\t", minNoInterCC, avgNoInterCC, maxNoInterCC);
    double avgCC =
      ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
    printf("%f\t%f\t%f\t",
      (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
      avgCC,
      (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));

    printf("%lu\t%lu\t", countInterCT, countNoInterCT);
    double avgInterCT = sumInterCT / (double)countInterCT;
    printf("%f\t%f\t%f\t", minInterCT, avgInterCT, maxInterCT);
    double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
    printf("%f\t%f\t%f\t", minNoInterCT, avgNoInterCT, maxNoInterCT);
    double avgCT =
      ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
    printf("%f\t%f\t%f\t",
      (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
      avgCT,
      (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));

    printf("%lu\t%lu\t", countInterTC, countNoInterTC);
    double avgInterTC = sumInterTC / (double)countInterTC;
    printf("%f\t%f\t%f\t", minInterTC, avgInterTC, maxInterTC);
    double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
    printf("%f\t%f\t%f\t", minNoInterTC, avgNoInterTC, maxNoInterTC);
    double avgTC =
      ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
    printf("%f\t%f\t%f\t",
      (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
      avgTC,
      (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));

    printf("%lu\t%lu\t", countInterTT, countNoInterTT);
    double avgInterTT = sumInterTT / (double)countInterTT;
    printf("%f\t%f\t%f\t", minInterTT, avgInterTT, maxInterTT);
    double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
    printf("%f\t%f\t%f\t", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
      ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    printf("%f\t%f\t%f\n",
      (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
      avgTT,
      (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));

  }

}

int main(int argc, char** argv) {

  Qualify3DStatic();

  return 0;
}
```

9.1.3 2D dynamic

```
// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
```

```c
#include <time.h>
#include <sys/time.h>

// Include FMB and SAT algorithm library
#include "fmb2dt.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 100000
// Nb of times the test is run on one pair of frame, used to
// slow down the processus and be able to measure time
#define NB_REPEAT_2D 1500

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Helper structure to pass arguments to the Qualification function
typedef struct {
  FrameType type;
  double orig[2];
  double comp[2][2];
  double speed[2];
} Param2DTime;

// Global variables to count nb of tests resulting in intersection
// and no intersection, and min/max/total time of execution for each
double minInter;
double maxInter;
double sumInter;
unsigned long countInter;
double minNoInter;
double maxNoInter;
double sumNoInter;
unsigned long countNoInter;

double minInterCC;
double maxInterCC;
double sumInterCC;
unsigned long countInterCC;
double minNoInterCC;
double maxNoInterCC;
double sumNoInterCC;
unsigned long countNoInterCC;

double minInterCT;
double maxInterCT;
double sumInterCT;
unsigned long countInterCT;
double minNoInterCT;
double maxNoInterCT;
double sumNoInterCT;
unsigned long countNoInterCT;

double minInterTC;
double maxInterTC;
double sumInterTC;
```

```
unsigned long countInterTC;
double minNoInterTC;
double maxNoInterTC;
double sumNoInterTC;
unsigned long countNoInterTC;

double minInterTT;
double maxInterTT;
double sumInterTT;
unsigned long countInterTT;
double minNoInterTT;
double maxNoInterTT;
double sumNoInterTT;
unsigned long countNoInterTT;

// Qualification function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and measure the time of execution of each
void Qualification2DDynamic(
        const Param2DTime paramP,
        const Param2DTime paramQ) {

  // Create the two Frames
  Frame2DTime P =
    Frame2DTimeCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.speed,
      paramP.comp);

  Frame2DTime Q =
    Frame2DTimeCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.speed,
      paramQ.comp);

  // Helper variables to loop on the pair (that, tho) and (tho, that)
  Frame2DTime* that = &P;
  Frame2DTime* tho = &Q;

  // Loop on pairs of Frames
  for (int iPair = 2;
       iPair--;) {

    // Declare an array to memorize the results of the repeated
    // test on the same pair,
    // to prevent optimization from the compiler to remove the for loop
    bool isIntersectingFMB[NB_REPEAT_2D] = {false};

    // Start measuring time
    struct timeval start;
    gettimeofday(&start, NULL);

    // Run the FMB intersection test
    for (int i = NB_REPEAT_2D;
         i--;) {

      isIntersectingFMB[i] =
        FMBTestIntersection2DTime(
          that,
          tho,
```

```
      NULL);
}

// Stop measuring time
struct timeval stop;
gettimeofday(&stop, NULL);

// Calculate the delay of execution
unsigned long deltausFMB = 0;
if (stop.tv_sec < start.tv_sec) {
  printf("time warps, try again\n");
  exit(0);
}
if (stop.tv_sec > start.tv_sec + 1) {
  printf("deltausFMB >> 1s, decrease NB_REPEAT\n");
  exit(0);
}
if (stop.tv_usec < start.tv_usec) {
  deltausFMB = stop.tv_sec - start.tv_sec;
  deltausFMB += stop.tv_usec + 1000000 - start.tv_usec;
} else {
  deltausFMB = stop.tv_usec - start.tv_usec;
}

// Declare an array to memorize the results of the repeated
// test on the same pair,
// to prevent optimization from the compiler to remove the for loop
bool isIntersectingSAT[NB_REPEAT_2D] = {false};

// Start measuring time
gettimeofday(&start, NULL);

// Run the FMB intersection test
for (int i = NB_REPEAT_2D;
     i--;) {

  isIntersectingSAT[i] =
    SATTestIntersection2DTime(
      that,
      tho);

}

// Stop measuring time
gettimeofday(&stop, NULL);

// Calculate the delay of execution
unsigned long deltausSAT = 0;
if (stop.tv_sec < start.tv_sec) {
  printf("time warps, try again\n");
  exit(0);
}
if (stop.tv_sec > start.tv_sec + 1) {
  printf("deltausSAT >> 1s, decrease NB_REPEAT\n");
  exit(0);
}
if (stop.tv_usec < start.tv_usec) {
  deltausSAT = stop.tv_sec - start.tv_sec;
  deltausSAT += stop.tv_usec + 1000000 - start.tv_usec;
} else {
  deltausSAT = stop.tv_usec - start.tv_usec;
}
```

```
// If the delays are greater than 10ms
if (deltausFMB >= 10 && deltausSAT >= 10) {

  // If FMB and SAT disagrees
  if (isIntersectingFMB[0] != isIntersectingSAT[0]) {

    printf("Qualification has failed\n");
    Frame2DTimePrint(that);
    printf(" against ");
    Frame2DTimePrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB[0] == false)
      printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT[0] == false)
      printf("no ");
    printf("intersection\n");

    // Stop the qualification test
    exit(0);

  }

  // Get the ratio of execution time
  double ratio = ((double)deltausFMB) / ((double)deltausSAT);

  // If the Frames intersect
  if (isIntersectingSAT[0] == true) {

    // Update the counters
    if (countInter == 0) {

      minInter = ratio;
      maxInter = ratio;

    } else {

      if (minInter > ratio)
        minInter = ratio;
      if (maxInter < ratio)
        maxInter = ratio;

    }
    sumInter += ratio;
    ++countInter;

    if (paramP.type == FrameCuboid &&
        paramQ.type == FrameCuboid) {

      if (countInterCC == 0) {

        minInterCC = ratio;
        maxInterCC = ratio;

      } else {

        if (minInterCC > ratio)
          minInterCC = ratio;
        if (maxInterCC < ratio)
```

```
      maxInterCC = ratio;

  }
  sumInterCC += ratio;
  ++countInterCC;

} else if (paramP.type == FrameCuboid &&
           paramQ.type == FrameTetrahedron) {

  if (countInterCT == 0) {

    minInterCT = ratio;
    maxInterCT = ratio;

  } else {

    if (minInterCT > ratio)
      minInterCT = ratio;
    if (maxInterCT < ratio)
      maxInterCT = ratio;

  }
  sumInterCT += ratio;
  ++countInterCT;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameCuboid) {

  if (countInterTC == 0) {

    minInterTC = ratio;
    maxInterTC = ratio;

  } else {

    if (minInterTC > ratio)
      minInterTC = ratio;
    if (maxInterTC < ratio)
      maxInterTC = ratio;

  }
  sumInterTC += ratio;
  ++countInterTC;

} else if (paramP.type == FrameTetrahedron &&
           paramQ.type == FrameTetrahedron) {

  if (countInterTT == 0) {

    minInterTT = ratio;
    maxInterTT = ratio;

  } else {

    if (minInterTT > ratio)
      minInterTT = ratio;
    if (maxInterTT < ratio)
      maxInterTT = ratio;

  }
  sumInterTT += ratio;
  ++countInterTT;
```

```
  }

// Else, the Frames do not intersect
} else {

  // Update the counters
  if (countNoInter == 0) {

    minNoInter = ratio;
    maxNoInter = ratio;

  } else {

    if (minNoInter > ratio)
      minNoInter = ratio;
    if (maxNoInter < ratio)
      maxNoInter = ratio;

  }
  sumNoInter += ratio;
  ++countNoInter;

  if (paramP.type == FrameCuboid &&
      paramQ.type == FrameCuboid) {

    if (countNoInterCC == 0) {

      minNoInterCC = ratio;
      maxNoInterCC = ratio;

    } else {

      if (minNoInterCC > ratio)
        minNoInterCC = ratio;
      if (maxNoInterCC < ratio)
        maxNoInterCC = ratio;

    }
    sumNoInterCC += ratio;
    ++countNoInterCC;

  } else if (paramP.type == FrameCuboid &&
             paramQ.type == FrameTetrahedron) {

    if (countNoInterCT == 0) {

      minNoInterCT = ratio;
      maxNoInterCT = ratio;

    } else {

      if (minNoInterCT > ratio)
        minNoInterCT = ratio;
      if (maxNoInterCT < ratio)
        maxNoInterCT = ratio;

    }
    sumNoInterCT += ratio;
    ++countNoInterCT;

  } else if (paramP.type == FrameTetrahedron &&
```

```
                 paramQ.type == FrameCuboid) {

          if (countNoInterTC == 0) {

            minNoInterTC = ratio;
            maxNoInterTC = ratio;

          } else {

            if (minNoInterTC > ratio)
              minNoInterTC = ratio;
            if (maxNoInterTC < ratio)
              maxNoInterTC = ratio;

          }
          sumNoInterTC += ratio;
          ++countNoInterTC;

        } else if (paramP.type == FrameTetrahedron &&
                   paramQ.type == FrameTetrahedron) {

          if (countNoInterTT == 0) {

            minNoInterTT = ratio;
            maxNoInterTT = ratio;

          } else {

            if (minNoInterTT > ratio)
              minNoInterTT = ratio;
            if (maxNoInterTT < ratio)
              maxNoInterTT = ratio;

          }
          sumNoInterTT += ratio;
          ++countNoInterTT;

        }
      }

    // Else, if time of execution for FMB was less than a 10ms
    } else if (deltausFMB < 10) {

      printf("deltausFMB < 10ms, increase NB_REPEAT\n");
      exit(0);

    // Else, if time of execution for SAT was less than a 10ms
    } else if (deltausSAT < 10) {

      printf("deltausSAT < 10ms, increase NB_REPEAT\n");
      exit(0);

    }

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

  }

}
```

```
void Qualify2DDynamic(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Loop on runs
  for (int iRun = 0;
       iRun < NB_RUNS;
       ++iRun) {

    // Ratio intersection/no intersection for the displayed results
    double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

    // Initialize counters
    minInter = 0.0;
    maxInter = 0.0;
    sumInter = 0.0;
    countInter = 0;
    minNoInter = 0.0;
    maxNoInter = 0.0;
    sumNoInter = 0.0;
    countNoInter = 0;

    minInterCC = 0.0;
    maxInterCC = 0.0;
    sumInterCC = 0.0;
    countInterCC = 0;
    minNoInterCC = 0.0;
    maxNoInterCC = 0.0;
    sumNoInterCC = 0.0;
    countNoInterCC = 0;

    minInterCT = 0.0;
    maxInterCT = 0.0;
    sumInterCT = 0.0;
    countInterCT = 0;
    minNoInterCT = 0.0;
    maxNoInterCT = 0.0;
    sumNoInterCT = 0.0;
    countNoInterCT = 0;

    minInterTC = 0.0;
    maxInterTC = 0.0;
    sumInterTC = 0.0;
    countInterTC = 0;
    minNoInterTC = 0.0;
    maxNoInterTC = 0.0;
    sumNoInterTC = 0.0;
    countNoInterTC = 0;

    minInterTT = 0.0;
    maxInterTT = 0.0;
    sumInterTT = 0.0;
    countInterTT = 0;
    minNoInterTT = 0.0;
    maxNoInterTT = 0.0;
    sumNoInterTT = 0.0;
    countNoInterTT = 0;

    // Declare two variables to memozie the arguments to the
    // Qualification function
    Param2DTime paramP;
```

```
Param2DTime paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

  // Create two random Frame definitions
  Param2DTime* param = &paramP;
  for (int iParam = 2;
       iParam--;) {

    // 50% chance of being a Cuboid or a Tetrahedron
    if (rnd() < 0.5)
      param->type = FrameCuboid;
    else
      param->type = FrameTetrahedron;

    for (int iAxis = 2;
         iAxis--;) {

      param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
      param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      for (int iComp = 2;
           iComp--;) {

        param->comp[iComp][iAxis] =
          -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      }

    }

    param = &paramQ;

  }

  // Calculate the determinant of the Frames' components matrix

  double detP =
    paramP.comp[0][0] * paramP.comp[1][1] -
    paramP.comp[1][0] * paramP.comp[0][1];

  double detQ =
    paramQ.comp[0][0] * paramQ.comp[1][1] -
    paramQ.comp[1][0] * paramQ.comp[0][1];

  // If the determinants are not null, ie the Frame are not degenerate
  if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Qualification2DDynamic(
      paramP,
      paramQ);

  }

}

// Display the results
if (iRun == 0) {
```

```
    printf("percPairInter\t");
    printf("countInter\tcountNoInter\t");
    printf("minInter\tavgInter\tmaxInter\t");
    printf("minNoInter\tavgNoInter\tmaxNoInter\t");
    printf("minTotal\tavgTotal\tmaxTotal\t");

    printf("countInterCC\tcountNoInterCC\t");
    printf("minInterCC\tavgInterCC\tmaxInterCC\t");
    printf("minNoInterCC\tavgNoInterCC\tmaxNoInterCC\t");
    printf("minTotalCC\tavgTotalCC\tmaxTotalCC\t");

    printf("countInterCT\tcountNoInterCT\t");
    printf("minInterCT\tavgInterCT\tmaxInterCT\t");
    printf("minNoInterCT\tavgNoInterCT\tmaxNoInterCT\t");
    printf("minTotalCT\tavgTotalCT\tmaxTotalCT\t");

    printf("countInterTC\tcountNoInterTC\t");
    printf("minInterTC\tavgInterTC\tmaxInterTC\t");
    printf("minNoInterTC\tavgNoInterTC\tmaxNoInterTC\t");
    printf("minTotalTC\tavgTotalTC\tmaxTotalTC\t");

    printf("countInterTT\tcountNoInterTT\t");
    printf("minInterTT\tavgInterTT\tmaxInterTT\t");
    printf("minNoInterTT\tavgNoInterTT\tmaxNoInterTT\t");
    printf("minTotalTT\tavgTotalTT\tmaxTotalTT\n");

  }

  printf("%.1f\t", ratioInter);

  printf("%lu\t%lu\t", countInter, countNoInter);
  double avgInter = sumInter / (double)countInter;
  printf("%f\t%f\t%f\t", minInter, avgInter, maxInter);
  double avgNoInter = sumNoInter / (double)countNoInter;
  printf("%f\t%f\t%f\t", minNoInter, avgNoInter, maxNoInter);
  double avg =
    ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
  printf("%f\t%f\t%f\t",
    (minNoInter < minInter ? minNoInter : minInter),
    avg,
    (maxNoInter > maxInter ? maxNoInter : maxInter));

  printf("%lu\t%lu\t", countInterCC, countNoInterCC);
  double avgInterCC = sumInterCC / (double)countInterCC;
  printf("%f\t%f\t%f\t", minInterCC, avgInterCC, maxInterCC);
  double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
  printf("%f\t%f\t%f\t", minNoInterCC, avgNoInterCC, maxNoInterCC);
  double avgCC =
    ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
  printf("%f\t%f\t%f\t",
    (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
    avgCC,
    (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));

  printf("%lu\t%lu\t", countInterCT, countNoInterCT);
  double avgInterCT = sumInterCT / (double)countInterCT;
  printf("%f\t%f\t%f\t", minInterCT, avgInterCT, maxInterCT);
  double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
  printf("%f\t%f\t%f\t", minNoInterCT, avgNoInterCT, maxNoInterCT);
  double avgCT =
    ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
  printf("%f\t%f\t%f\t",
```

```
      (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
      avgCT,
      (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));

    printf("%lu\t%lu\t", countInterTC, countNoInterTC);
    double avgInterTC = sumInterTC / (double)countInterTC;
    printf("%f\t%f\t%f\t", minInterTC, avgInterTC, maxInterTC);
    double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
    printf("%f\t%f\t%f\t", minNoInterTC, avgNoInterTC, maxNoInterTC);
    double avgTC =
      ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
    printf("%f\t%f\t%f\t",
      (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
      avgTC,
      (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));

    printf("%lu\t%lu\t", countInterTT, countNoInterTT);
    double avgInterTT = sumInterTT / (double)countInterTT;
    printf("%f\t%f\t%f\t", minInterTT, avgInterTT, maxInterTT);
    double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
    printf("%f\t%f\t%f\t", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
      ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    printf("%f\t%f\t%f\n",
      (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
      avgTT,
      (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));

  }

}

int main(int argc, char** argv) {

  Qualify2DDynamic();

  return 0;
}
```

9.1.4 3D dynamic

```
// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>

// Include FMB and SAT algorithm library
#include "fmb3dt.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 100000
// Nb of times the test is run on one pair of frame, used to
```

```
// slow down the processus and be able to measure time
#define NB_REPEAT_3D 800

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Helper structure to pass arguments to the Qualification function
typedef struct {
  FrameType type;
  double orig[3];
  double comp[3][3];
  double speed[3];
} Param3DTime;

// Global variables to count nb of tests resulting in intersection
// and no intersection, and min/max/total time of execution for each
double minInter;
double maxInter;
double sumInter;
unsigned long countInter;
double minNoInter;
double maxNoInter;
double sumNoInter;
unsigned long countNoInter;

double minInterCC;
double maxInterCC;
double sumInterCC;
unsigned long countInterCC;
double minNoInterCC;
double maxNoInterCC;
double sumNoInterCC;
unsigned long countNoInterCC;

double minInterCT;
double maxInterCT;
double sumInterCT;
unsigned long countInterCT;
double minNoInterCT;
double maxNoInterCT;
double sumNoInterCT;
unsigned long countNoInterCT;

double minInterTC;
double maxInterTC;
double sumInterTC;
unsigned long countInterTC;
double minNoInterTC;
double maxNoInterTC;
double sumNoInterTC;
unsigned long countNoInterTC;

double minInterTT;
double maxInterTT;
double sumInterTT;
unsigned long countInterTT;
double minNoInterTT;
double maxNoInterTT;
double sumNoInterTT;
unsigned long countNoInterTT;

// Qualification function
```

```
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and measure the time of execution of each
void Qualification3DDynamic(
        const Param3DTime paramP,
        const Param3DTime paramQ) {

  // Create the two Frames
  Frame3DTime P =
    Frame3DTimeCreateStatic(
      paramP.type,
      paramP.orig,
      paramP.speed,
      paramP.comp);

  Frame3DTime Q =
    Frame3DTimeCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.speed,
      paramQ.comp);

  // Helper variables to loop on the pair (that, tho) and (tho, that)
  Frame3DTime* that = &P;
  Frame3DTime* tho = &Q;

  // Loop on pairs of Frames
  for (int iPair = 2;
       iPair--;) {

    // Declare an array to memorize the results of the repeated
    // test on the same pair,
    // to prevent optimization from the compiler to remove the for loop
    bool isIntersectingFMB[NB_REPEAT_3D] = {false};

    // Start measuring time
    struct timeval start;
    gettimeofday(&start, NULL);

    // Run the FMB intersection test
    for (int i = NB_REPEAT_3D;
         i--;) {

      isIntersectingFMB[i] =
        FMBTestIntersection3DTime(
          that,
          tho,
          NULL);
    }

    // Stop measuring time
    struct timeval stop;
    gettimeofday(&stop, NULL);

    // Calculate the delay of execution
    unsigned long deltausFMB = 0;
    if (stop.tv_sec < start.tv_sec) {
      printf("time warps, try again\n");
      exit(0);
    }
    if (stop.tv_sec > start.tv_sec + 1) {
      printf("deltausFMB >> 1s, decrease NB_REPEAT\n");
      exit(0);
```

```
}
if (stop.tv_usec < start.tv_usec) {
  deltausFMB = stop.tv_sec - start.tv_sec;
  deltausFMB += stop.tv_usec + 1000000 - start.tv_usec;
} else {
  deltausFMB = stop.tv_usec - start.tv_usec;
}

// Declare an array to memorize the results of the repeated
// test on the same pair,
// to prevent optimization from the compiler to remove the for loop
bool isIntersectingSAT[NB_REPEAT_3D] = {false};

// Start measuring time
gettimeofday(&start, NULL);

// Run the FMB intersection test
for (int i = NB_REPEAT_3D;
     i--;) {

  isIntersectingSAT[i] =
    SATTestIntersection3DTime(
      that,
      tho);

}

// Stop measuring time
gettimeofday(&stop, NULL);

// Calculate the delay of execution
unsigned long deltausSAT = 0;
if (stop.tv_sec < start.tv_sec) {
  printf("time warps, try again\n");
  exit(0);
}
if (stop.tv_sec > start.tv_sec + 1) {
  printf("deltausSAT >> 1s, decrease NB_REPEAT\n");
  exit(0);
}
if (stop.tv_usec < start.tv_usec) {
  deltausSAT = stop.tv_sec - start.tv_sec;
  deltausSAT += stop.tv_usec + 1000000 - start.tv_usec;
} else {
  deltausSAT = stop.tv_usec - start.tv_usec;
}

// If the delays are greater than 10ms
if (deltausFMB >= 10 && deltausSAT >= 10) {

  // If FMB and SAT disagrees
  if (isIntersectingFMB[0] != isIntersectingSAT[0]) {

    printf("Qualification has failed\n");
    Frame3DTimePrint(that);
    printf(" against ");
    Frame3DTimePrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB[0] == false)
      printf("no ");
    printf("intersection\n");
```

```
  printf("SAT : ");
  if (isIntersectingSAT[0] == false)
    printf("no ");
  printf("intersection\n");

  // Stop the qualification test
  exit(0);

}

// Get the ratio of execution time
double ratio = ((double)deltausFMB) / ((double)deltausSAT);

// If the Frames intersect
if (isIntersectingSAT[0] == true) {

  // Update the counters
  if (countInter == 0) {

    minInter = ratio;
    maxInter = ratio;

  } else {

    if (minInter > ratio)
      minInter = ratio;
    if (maxInter < ratio)
      maxInter = ratio;

  }
  sumInter += ratio;
  ++countInter;

  if (paramP.type == FrameCuboid &&
      paramQ.type == FrameCuboid) {

    if (countInterCC == 0) {

      minInterCC = ratio;
      maxInterCC = ratio;

    } else {

      if (minInterCC > ratio)
        minInterCC = ratio;
      if (maxInterCC < ratio)
        maxInterCC = ratio;

    }
    sumInterCC += ratio;
    ++countInterCC;

  } else if (paramP.type == FrameCuboid &&
             paramQ.type == FrameTetrahedron) {

    if (countInterCT == 0) {

      minInterCT = ratio;
      maxInterCT = ratio;

    } else {
```

```
      if (minInterCT > ratio)
        minInterCT = ratio;
      if (maxInterCT < ratio)
        maxInterCT = ratio;

    }
    sumInterCT += ratio;
    ++countInterCT;

  } else if (paramP.type == FrameTetrahedron &&
             paramQ.type == FrameCuboid) {

    if (countInterTC == 0) {

      minInterTC = ratio;
      maxInterTC = ratio;

    } else {

      if (minInterTC > ratio)
        minInterTC = ratio;
      if (maxInterTC < ratio)
        maxInterTC = ratio;

    }
    sumInterTC += ratio;
    ++countInterTC;

  } else if (paramP.type == FrameTetrahedron &&
             paramQ.type == FrameTetrahedron) {

    if (countInterTT == 0) {

      minInterTT = ratio;
      maxInterTT = ratio;

    } else {

      if (minInterTT > ratio)
        minInterTT = ratio;
      if (maxInterTT < ratio)
        maxInterTT = ratio;

    }
    sumInterTT += ratio;
    ++countInterTT;

  }

// Else, the Frames do not intersect
} else {

  // Update the counters
  if (countNoInter == 0) {

    minNoInter = ratio;
    maxNoInter = ratio;

  } else {

    if (minNoInter > ratio)
      minNoInter = ratio;
```

```
    if (maxNoInter < ratio)
      maxNoInter = ratio;

  }
  sumNoInter += ratio;
  ++countNoInter;

  if (paramP.type == FrameCuboid &&
      paramQ.type == FrameCuboid) {

    if (countNoInterCC == 0) {

      minNoInterCC = ratio;
      maxNoInterCC = ratio;

    } else {

      if (minNoInterCC > ratio)
        minNoInterCC = ratio;
      if (maxNoInterCC < ratio)
        maxNoInterCC = ratio;

    }
    sumNoInterCC += ratio;
    ++countNoInterCC;

  } else if (paramP.type == FrameCuboid &&
             paramQ.type == FrameTetrahedron) {

    if (countNoInterCT == 0) {

      minNoInterCT = ratio;
      maxNoInterCT = ratio;

    } else {

      if (minNoInterCT > ratio)
        minNoInterCT = ratio;
      if (maxNoInterCT < ratio)
        maxNoInterCT = ratio;

    }
    sumNoInterCT += ratio;
    ++countNoInterCT;

  } else if (paramP.type == FrameTetrahedron &&
             paramQ.type == FrameCuboid) {

    if (countNoInterTC == 0) {

      minNoInterTC = ratio;
      maxNoInterTC = ratio;

    } else {

      if (minNoInterTC > ratio)
        minNoInterTC = ratio;
      if (maxNoInterTC < ratio)
        maxNoInterTC = ratio;

    }
    sumNoInterTC += ratio;
```

```
          ++countNoInterTC;

        } else if (paramP.type == FrameTetrahedron &&
                   paramQ.type == FrameTetrahedron) {

          if (countNoInterTT == 0) {

            minNoInterTT = ratio;
            maxNoInterTT = ratio;

          } else {

            if (minNoInterTT > ratio)
              minNoInterTT = ratio;
            if (maxNoInterTT < ratio)
              maxNoInterTT = ratio;

          }
          sumNoInterTT += ratio;
          ++countNoInterTT;

        }
      }

    // Else, if time of execution for FMB was less than a 10ms
    } else if (deltausFMB < 10) {

      printf("deltausFMB < 10ms, increase NB_REPEAT\n");
      exit(0);

    // Else, if time of execution for SAT was less than a 10ms
    } else if (deltausSAT < 10) {

      printf("deltausSAT < 10ms, increase NB_REPEAT\n");
      exit(0);

    }

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

  }

}

void Qualify3DDynamic(void) {

  // Initialise the random generator
  srandom(time(NULL));

  // Loop on runs
  for (int iRun = 0;
       iRun < NB_RUNS;
       ++iRun) {

    // Ratio intersection/no intersection for the displayed results
    double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

    // Initialize counters
    minInter = 0.0;
    maxInter = 0.0;
```

```
sumInter = 0.0;
countInter = 0;
minNoInter = 0.0;
maxNoInter = 0.0;
sumNoInter = 0.0;
countNoInter = 0;

minInterCC = 0.0;
maxInterCC = 0.0;
sumInterCC = 0.0;
countInterCC = 0;
minNoInterCC = 0.0;
maxNoInterCC = 0.0;
sumNoInterCC = 0.0;
countNoInterCC = 0;

minInterCT = 0.0;
maxInterCT = 0.0;
sumInterCT = 0.0;
countInterCT = 0;
minNoInterCT = 0.0;
maxNoInterCT = 0.0;
sumNoInterCT = 0.0;
countNoInterCT = 0;

minInterTC = 0.0;
maxInterTC = 0.0;
sumInterTC = 0.0;
countInterTC = 0;
minNoInterTC = 0.0;
maxNoInterTC = 0.0;
sumNoInterTC = 0.0;
countNoInterTC = 0;

minInterTT = 0.0;
maxInterTT = 0.0;
sumInterTT = 0.0;
countInterTT = 0;
minNoInterTT = 0.0;
maxNoInterTT = 0.0;
sumNoInterTT = 0.0;
countNoInterTT = 0;

// Declare two variables to memozie the arguments to the
// Qualification function
Param3DTime paramP;
Param3DTime paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

  // Create two random Frame definitions
  Param3DTime* param = &paramP;
  for (int iParam = 2;
       iParam--;) {

    // 50% chance of being a Cuboid or a Tetrahedron
    if (rnd() < 0.5)
      param->type = FrameCuboid;
    else
      param->type = FrameTetrahedron;
```

```
    for (int iAxis = 3;
         iAxis--;) {

      param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
      param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      for (int iComp = 3;
           iComp--;) {

        param->comp[iComp][iAxis] =
          -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

      }

    }

    param = &paramQ;

  }

  // Calculate the determinant of the Frames' components matrix

double detP =
  paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2]-
  paramP.comp[1][2] * paramP.comp[2][1]) -
  paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2]-
  paramP.comp[0][2] * paramP.comp[2][1]) +
  paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2]-
  paramP.comp[0][2] * paramP.comp[1][1]);

double detQ =
  paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2]-
  paramQ.comp[1][2] * paramQ.comp[2][1]) -
  paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2]-
  paramQ.comp[0][2] * paramQ.comp[2][1]) +
  paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2]-
  paramQ.comp[0][2] * paramQ.comp[1][1]);

  // If the determinants are not null, ie the Frame are not degenerate
  if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Qualification3DDynamic(
      paramP,
      paramQ);

  }

}

// Display the results
if (iRun == 0) {

  printf("percPairInter\t");
  printf("countInter\tcountNoInter\t");
  printf("minInter\tavgInter\tmaxInter\t");
  printf("minNoInter\tavgNoInter\tmaxNoInter\t");
  printf("minTotal\tavgTotal\tmaxTotal\t");

  printf("countInterCC\tcountNoInterCC\t");
  printf("minInterCC\tavgInterCC\tmaxInterCC\t");
```

```
  printf("minNoInterCC\tavgNoInterCC\tmaxNoInterCC\t");
  printf("minTotalCC\tavgTotalCC\tmaxTotalCC\t");

  printf("countInterCT\tcountNoInterCT\t");
  printf("minInterCT\tavgInterCT\tmaxInterCT\t");
  printf("minNoInterCT\tavgNoInterCT\tmaxNoInterCT\t");
  printf("minTotalCT\tavgTotalCT\tmaxTotalCT\t");

  printf("countInterTC\tcountNoInterTC\t");
  printf("minInterTC\tavgInterTC\tmaxInterTC\t");
  printf("minNoInterTC\tavgNoInterTC\tmaxNoInterTC\t");
  printf("minTotalTC\tavgTotalTC\tmaxTotalTC\t");

  printf("countInterTT\tcountNoInterTT\t");
  printf("minInterTT\tavgInterTT\tmaxInterTT\t");
  printf("minNoInterTT\tavgNoInterTT\tmaxNoInterTT\t");
  printf("minTotalTT\tavgTotalTT\tmaxTotalTT\n");

}

printf("%.1f\t", ratioInter);

printf("%lu\t%lu\t", countInter, countNoInter);
double avgInter = sumInter / (double)countInter;
printf("%f\t%f\t%f\t", minInter, avgInter, maxInter);
double avgNoInter = sumNoInter / (double)countNoInter;
printf("%f\t%f\t%f\t", minNoInter, avgNoInter, maxNoInter);
double avg =
  ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
printf("%f\t%f\t%f\t",
  (minNoInter < minInter ? minNoInter : minInter),
  avg,
  (maxNoInter > maxInter ? maxNoInter : maxInter));

printf("%lu\t%lu\t", countInterCC, countNoInterCC);
double avgInterCC = sumInterCC / (double)countInterCC;
printf("%f\t%f\t%f\t", minInterCC, avgInterCC, maxInterCC);
double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
printf("%f\t%f\t%f\t", minNoInterCC, avgNoInterCC, maxNoInterCC);
double avgCC =
  ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
printf("%f\t%f\t%f\t",
  (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
  avgCC,
  (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));

printf("%lu\t%lu\t", countInterCT, countNoInterCT);
double avgInterCT = sumInterCT / (double)countInterCT;
printf("%f\t%f\t%f\t", minInterCT, avgInterCT, maxInterCT);
double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
printf("%f\t%f\t%f\t", minNoInterCT, avgNoInterCT, maxNoInterCT);
double avgCT =
  ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
printf("%f\t%f\t%f\t",
  (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
  avgCT,
  (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));

printf("%lu\t%lu\t", countInterTC, countNoInterTC);
double avgInterTC = sumInterTC / (double)countInterTC;
printf("%f\t%f\t%f\t", minInterTC, avgInterTC, maxInterTC);
double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
```

```
    printf("%f\t%f\t%f\t", minNoInterTC, avgNoInterTC, maxNoInterTC);
    double avgTC =
      ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
    printf("%f\t%f\t%f\t",
      (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
      avgTC,
      (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));

    printf("%lu\t%lu\t", countInterTT, countNoInterTT);
    double avgInterTT = sumInterTT / (double)countInterTT;
    printf("%f\t%f\t%f\t", minInterTT, avgInterTT, maxInterTT);
    double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
    printf("%f\t%f\t%f\t", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
      ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    printf("%f\t%f\t%f\n",
      (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
      avgTT,
      (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));

  }

}

int main(int argc, char** argv) {

  Qualify3DDynamic();

  return 0;
}
```

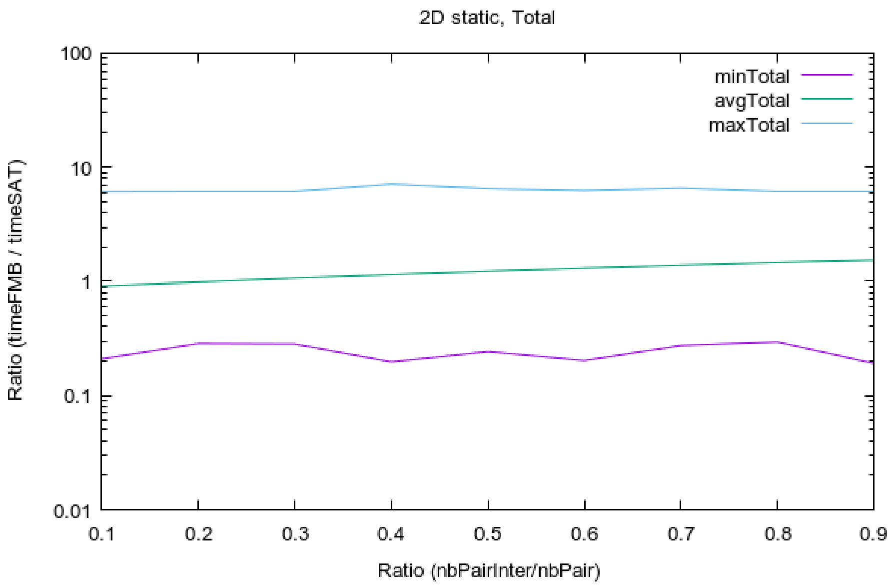
9.2 Results

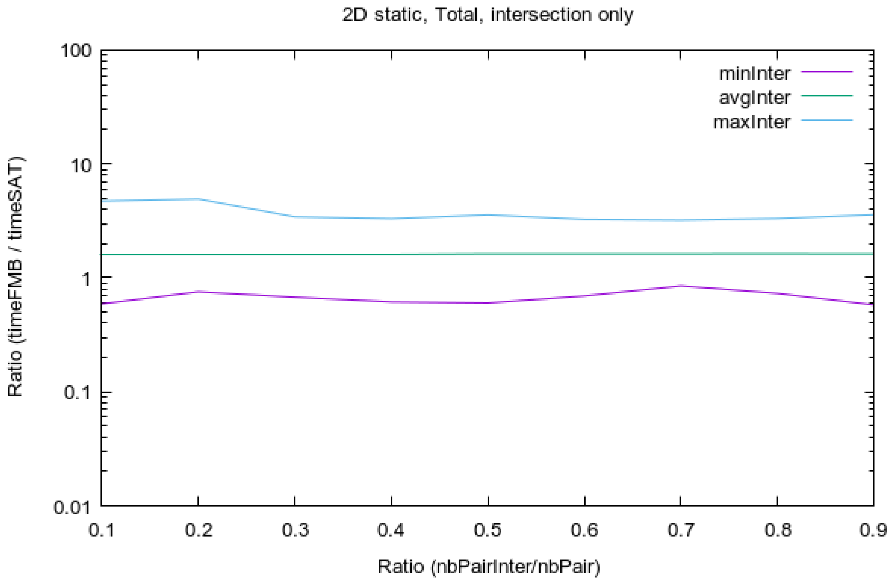
9.2.1 2D static

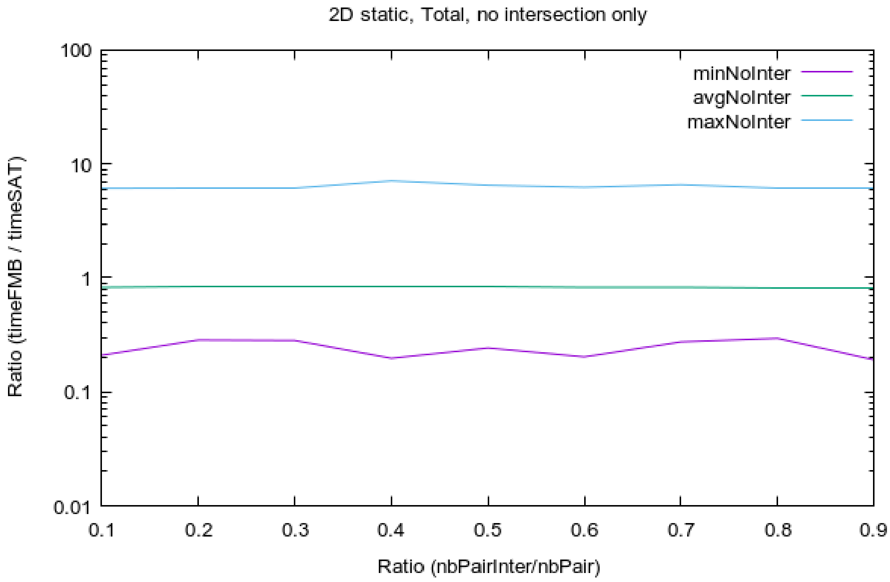
```
percPairInter   countInter      countNoInter    minInter        avgInter
            maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    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     46614   153384  0.591716        1.619298        4.722222
    0.209677        0.830309        6.133333        0.209677        0.909208
            6.133333        13076   36708   1.000000        2.091329
    4.722222        0.222222        0.754368        3.627907        0.222222
            0.888064        4.722222        11634   38902   0.883495
    1.574429        2.512821        0.325000        0.835568        6.133333
            0.325000        0.909454        6.133333        11692   38130
    0.591716        1.574867        2.338983        0.209677        0.840402
            6.133333        0.209677        0.913849        6.133333
    10212   39644   0.723810        1.116871        1.859375        0.245283
            0.885757        5.785714        0.245283        0.908868
    5.785714
```

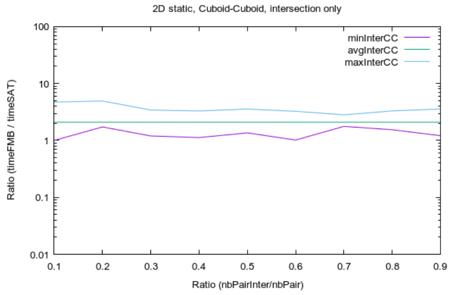
```
0.2     46388   153598  0.757009        1.619703        4.929825
    0.285714        0.830665        6.200000        0.285714        0.988472
            6.200000        13010   37094   1.727273        2.092507
    4.929825        0.361111        0.750883        3.000000        0.361111
            1.019208        4.929825        11462   38326   1.281690
    1.575129        1.950820        0.285714        0.835930        6.200000
            0.285714        0.983770        6.200000        11722   38544
    1.289474        1.575225        2.786885        0.309524        0.842047
            6.133333        0.309524        0.988682        6.133333
    10194   39634   0.757009        1.117555        1.695652        0.300000
            0.889173        5.642857        0.300000        0.934849
    5.642857
0.3     46784   153206  0.677686        1.619927        3.425926
    0.283019        0.830575        6.200000        0.283019        1.067381
            6.200000        13012   36728   1.200000        2.091625
    3.425926        0.283019        0.752638        3.025641        0.283019
            1.154334        3.425926        11780   38592   0.920792
    1.575296        2.271186        0.333333        0.842207        6.200000
            0.333333        1.062134        6.200000        11854   38248
    1.096386        1.576111        2.196721        0.302326        0.841250
            6.133333        0.302326        1.061708        6.133333
    10138   39638   0.677686        1.117600        1.695652        0.319444
            0.881165        5.642857        0.319444        0.952095
    5.642857
0.4     46602   153390  0.616000        1.619226        3.296296
    0.196970        0.831406        7.133333        0.196970        1.146534
            7.133333        13124   37014   1.123810        2.091809
    3.296296        0.196970        0.751607        4.102564        0.196970
            1.287688        4.102564        11604   38062   0.925926
    1.574914        2.606557        0.309524        0.839946        6.133333
            0.309524        1.133933        6.133333        11556   38528
    0.978947        1.575160        2.311475        0.215385        0.845174
            6.133333        0.215385        1.137168        6.133333
    10318   39786   0.616000        1.117313        1.985294        0.228070
            0.884144        7.133333        0.228070        0.977411
    7.133333
0.5     46868   153124  0.603175        1.621543        3.561644
    0.242857        0.830581        6.533333        0.242857        1.226062
            6.533333        13254   36406   1.357143        2.091315
    3.561644        0.371429        0.752506        3.050000        0.371429
            1.421911        3.561644        11818   38240   0.859813
    1.575325        2.483333        0.242857        0.841244        6.200000
            0.242857        1.208284        6.200000        11602   38768
    1.052632        1.575940        2.180328        0.317073        0.838673
            6.266667        0.317073        1.207307        6.266667
    10194   39710   0.603175        1.116239        1.400000        0.319444
            0.883992        6.533333        0.319444        1.000115
    6.533333
0.6     46602   153386  0.696429        1.619668        3.250000
    0.202899        0.830363        6.266667        0.202899        1.303946
            6.266667        13088   37200   1.017391        2.092002
    3.250000        0.202899        0.748680        3.025641        0.202899
            1.554673        3.250000        11632   37846   0.947917
    1.574833        2.400000        0.207547        0.841342        6.200000
            0.207547        1.281437        6.200000        11654   38642
    0.968750        1.575482        2.262295        0.207547        0.845625
            6.266667        0.207547        1.283539        6.266667
    10228   39698   0.696429        1.116594        1.671429        0.254237
            0.881581        5.571429        0.254237        1.022589
    5.571429
0.7     47224   152766  0.854545        1.620245        3.203390
    0.275000        0.830935        6.600000        0.275000        1.383452
```

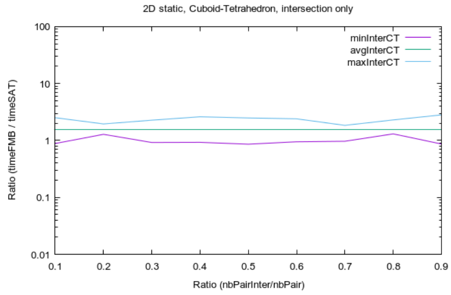
```
          6.600000        13312   36532   1.765625        2.091469
     2.814815        0.275000        0.752037        3.000000        0.275000
          1.689640        3.000000        11848   38030   0.968421
     1.574567        1.841270        0.280702        0.837227        6.266667
          0.280702        1.353365        6.266667        11714   38874
     0.854545        1.575705        3.203390        0.288889        0.843822
          6.600000        0.288889        1.356140        6.600000
     10350   39330   0.987342        1.116861        1.362319        0.317073
          0.885399        6.333333        0.317073        1.047422
     6.333333
0.8     46882   153114  0.733333        1.624254        3.307692
     0.295455        0.829368        6.133333        0.295455        1.465277
          6.133333        13384   37080   1.540541        2.092324
     3.307692        0.351351        0.752108        3.571429        0.351351
          1.824281        3.571429        11874   38278   1.304348
     1.574749        2.293103        0.295455        0.838582        6.133333
          0.295455        1.427516        6.133333        11554   37990
     0.912281        1.575525        2.196721        0.333333        0.837791
          6.133333        0.333333        1.427978        6.133333
     10070   39766   0.733333        1.116426        1.242424        0.319444
          0.884494        5.571429        0.319444        1.070039
     5.571429
0.9     46600   153392  0.579235        1.619221        3.574074
     0.191176        0.829271        6.133333        0.191176        1.540226
          6.133333        13118   37040   1.212766        2.091051
     3.574074        0.196970        0.750799        3.666667        0.196970
          1.957026        3.666667        11718   38520   0.866667
     1.574899        2.810345        0.244444        0.838097        6.133333
          0.244444        1.501219        6.133333        11486   37916
     0.857143        1.574810        2.862069        0.191176        0.839483
          6.133333        0.191176        1.501277        6.133333
     10278   39916   0.579235        1.117176        1.791045        0.209677
          0.883873        5.571429        0.209677        1.093846
     5.571429
```

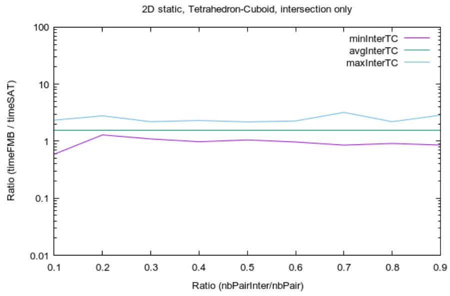


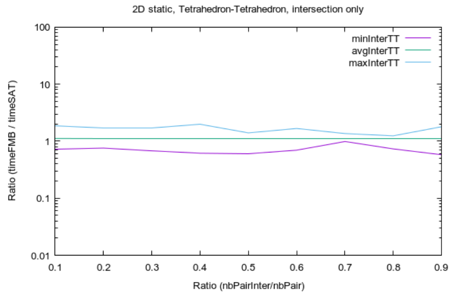

2D static, Total, intersection only
minInter
avgInter
maxInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

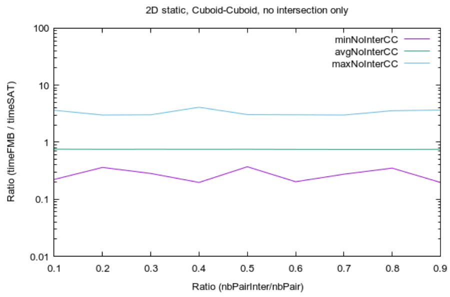

2D static, Total, no intersection only
minNoInter
avgNoInter
maxNoInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

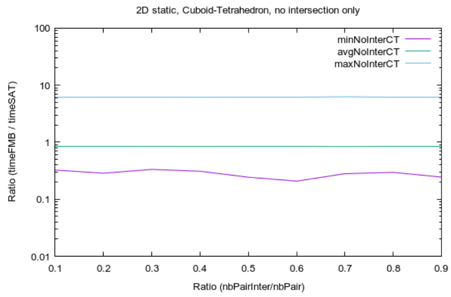


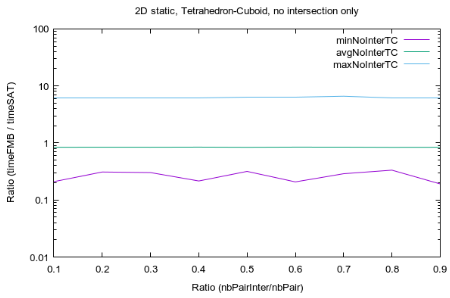


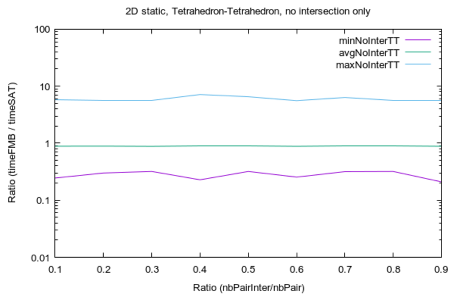








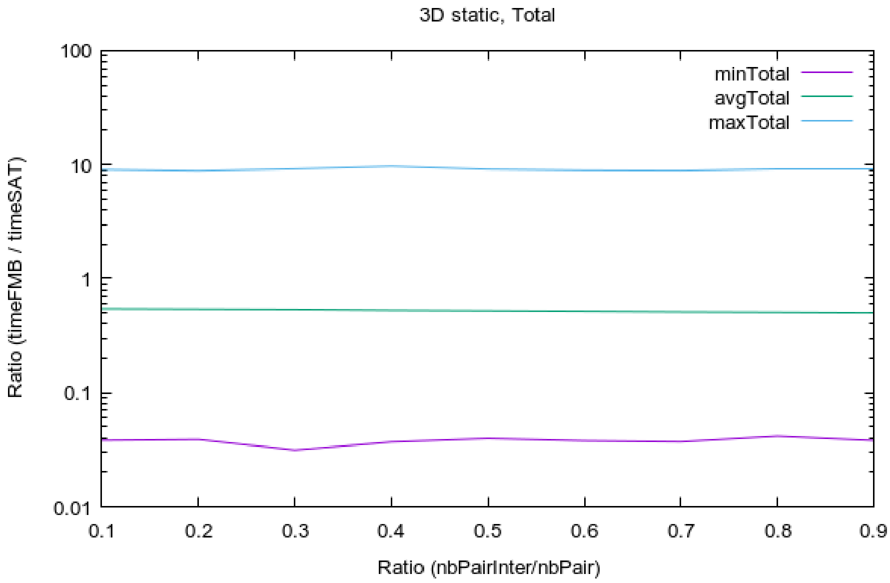


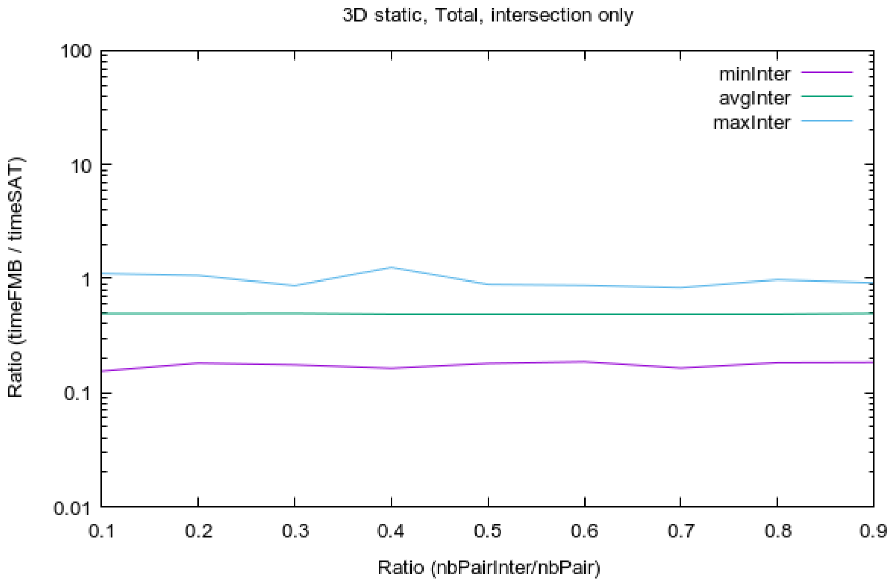


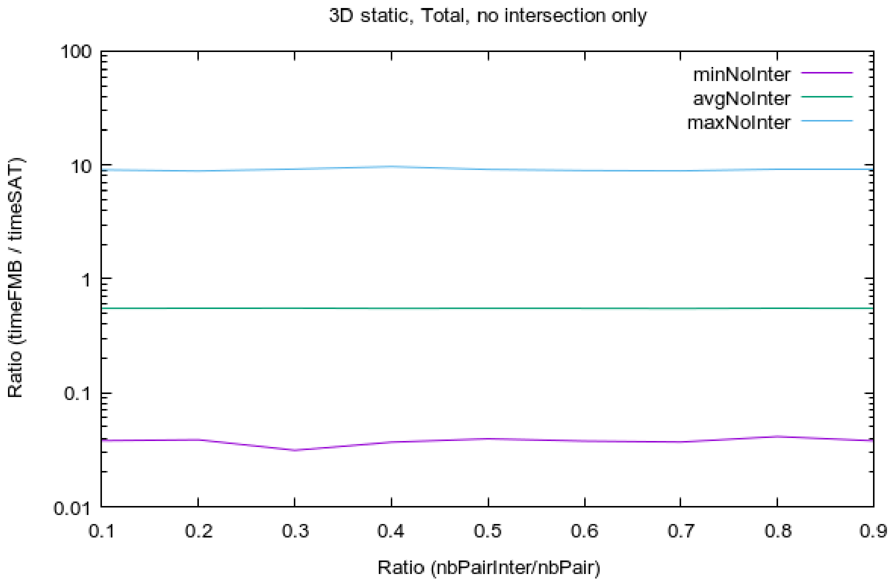
9.2.2 3D static

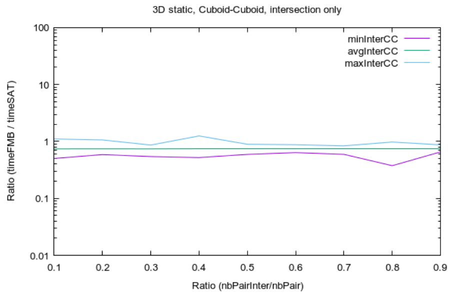
```
percPairInter   countInter      countNoInter    minInter        avgInter
        maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    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     31478   168522  0.153967        0.494154        1.104478
    0.038082        0.548169        9.062500        0.038082        0.542768
        9.062500        10616   39180   0.502475        0.741534
    1.104478        0.063596        0.395844        2.951049        0.063596
        0.430413        2.951049        7858    42396   0.224467
    0.415367        0.725434        0.046053        0.536495        9.062500
        0.046053        0.524382        9.062500        7822    42292
    0.222951        0.415492        0.679012        0.045171        0.531304
        8.281250        0.045171        0.519723        8.281250
    5182    44654   0.153967        0.225571        0.292978        0.038082
        0.708880        7.920000        0.038082        0.660549
    7.920000
0.2     31990   168008  0.181401        0.493077        1.062958
    0.038793        0.549532        8.843750        0.038793        0.538241
        8.843750        10716   39062   0.587977        0.741692
    1.062958        0.062762        0.397876        2.965517        0.062762
        0.466639        2.965517        7820    42262   0.274045
    0.415494        0.577810        0.045902        0.537255        8.750000
        0.045902        0.512903        8.750000        8100    42214
    0.271622        0.415845        0.581197        0.045381        0.532207
        8.843750        0.045381        0.508934        8.843750
    5354    44470   0.181401        0.225637        0.296252        0.038793
        0.710857        8.416667        0.038793        0.613813
    8.416667
0.3     32034   167966  0.175223        0.494816        0.871870
    0.031298        0.551022        9.200000        0.031298        0.534160
        9.200000        10930   39572   0.542582        0.741431
    0.864407        0.065646        0.395645        2.944828        0.065646
        0.499381        2.944828        7728    41900   0.274968
    0.415915        0.871870        0.045161        0.537350        9.200000
        0.045161        0.500919        9.200000        7960    41816
    0.272230        0.415764        0.570381        0.045016        0.534694
        8.968750        0.045016        0.499015        8.968750
    5416    44678   0.175223        0.225889        0.279213        0.031298
        0.716748        8.125000        0.031298        0.569490
    8.125000
0.4     31818   168182  0.163245        0.493042        1.250760
    0.036936        0.547619        9.677419        0.036936        0.525788
        9.677419        10616   39210   0.521236        0.741773
    1.250760        0.062500        0.395155        3.027397        0.062500
        0.533802        3.027397        7940    41910   0.274590
    0.415556        0.572464        0.045677        0.529730        9.677419
        0.045677        0.484060        9.677419        8000    42360
    0.271429        0.415905        0.563609        0.045234        0.528124
        8.937500        0.045234        0.483236        8.937500
    5262    44702   0.163245        0.225429        0.404678        0.036936
        0.716598        8.166667        0.036936        0.520130
    8.166667
```

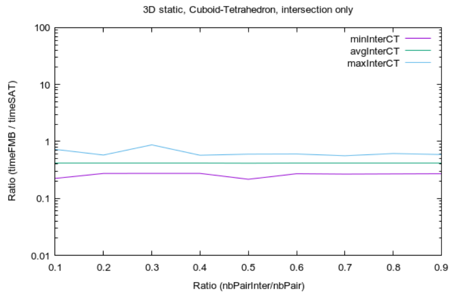
```
0.5     31848   168152  0.180208        0.493484        0.892791
    0.039548        0.549296        9.129032        0.039548        0.521390
            9.129032        10718   39586   0.592424        0.741641
    0.892791        0.066514        0.396111        2.931034        0.066514
            0.568876        2.931034        7814    41986   0.216713
    0.415310        0.598837        0.044776        0.531125        9.129032
            0.044776        0.473217        9.129032        7984    41724
    0.261034        0.415648        0.577424        0.046667        0.532189
            9.064516        0.046667        0.473918        9.064516
    5332    44856   0.180208        0.225771        0.287860        0.039548
            0.717407        8.565217        0.039548        0.471589
    8.565217
0.6     31236   168764  0.186603        0.493514        0.877323
    0.037762        0.548509        8.937500        0.037762        0.515512
            8.937500        10400   39442   0.637048        0.741701
    0.877323        0.064302        0.394465        3.157534        0.064302
            0.602807        3.157534        7782    42014   0.271255
    0.415913        0.602339        0.042945        0.531548        8.774194
            0.042945        0.462167        8.774194        7988    42142
    0.267568        0.415687        0.568047        0.045752        0.533325
            8.937500        0.045752        0.462742        8.937500
    5066    45166   0.186603        0.225930        0.316877        0.037762
            0.712975        8.166667        0.037762        0.420748
    8.166667
0.7     31100   168898  0.164050        0.493901        0.837887
    0.037037        0.546780        8.875000        0.037037        0.509765
            8.875000        10416   39924   0.593607        0.741501
    0.837887        0.066059        0.398390        3.000000        0.066059
            0.638568        3.000000        7764    42496   0.267023
    0.415769        0.560870        0.045307        0.534521        8.718750
            0.045307        0.451395        8.718750        7866    42116
    0.256989        0.415539        0.637037        0.045597        0.527322
            8.875000        0.045597        0.449074        8.875000
    5054    44362   0.164050        0.225602        0.407494        0.037037
            0.710542        8.375000        0.037037        0.371084
    8.375000
0.8     31570   168430  0.183099        0.494058        0.977570
    0.041379        0.550069        9.129032        0.041379        0.505260
            9.129032        10658   38766   0.374885        0.741284
    0.977570        0.067285        0.393096        3.113772        0.067285
            0.671646        3.113772        7898    42034   0.269076
    0.415545        0.612991        0.046205        0.536427        9.031250
            0.046205        0.439721        9.031250        7774    42434
    0.264892        0.415891        0.572474        0.046589        0.531436
            9.129032        0.046589        0.439000        9.129032
    5240    45196   0.183099        0.225512        0.304348        0.041379
            0.714891        8.208333        0.041379        0.323388
    8.208333
0.9     31536   168464  0.184149        0.494905        0.919591
    0.037975        0.548227        9.129032        0.037975        0.500237
            9.129032        10612   39240   0.654605        0.741446
    0.871841        0.065611        0.397763        2.917241        0.065611
            0.707078        2.917241        7876    42208   0.271255
    0.415875        0.589706        0.045902        0.535402        8.906250
            0.045902        0.427827        8.906250        7976    41912
    0.252488        0.416167        0.919591        0.045814        0.530613
            9.129032        0.045814        0.427611        9.129032
    5072    45104   0.184149        0.225615        0.273973        0.037975
            0.707499        8.120000        0.037975        0.273803
    8.120000
```

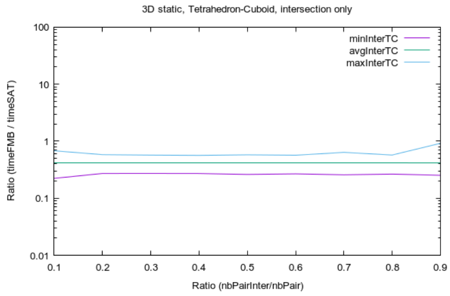

3D static, Total
minTotal
avgTotal
maxTotal
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

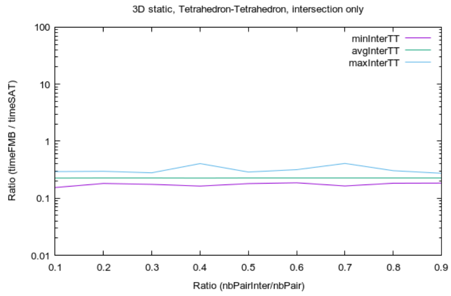

3D static, Total, intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minInter
avgInter
maxInter
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

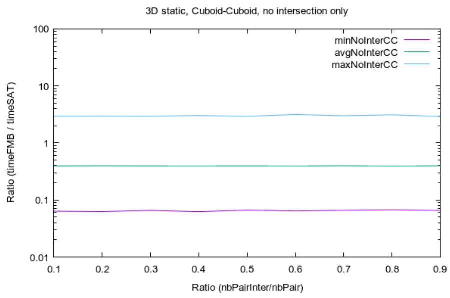

3D static, Total, no intersection only
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
minNoInter
avgNoInter
maxNoInter
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

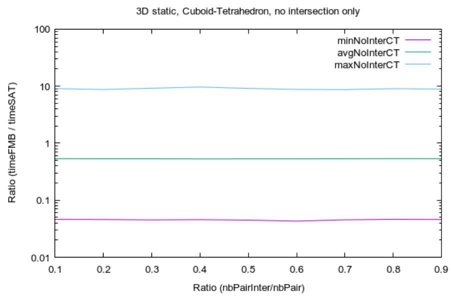


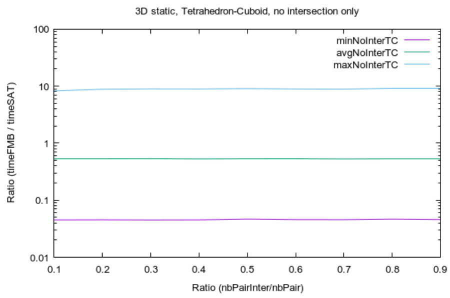


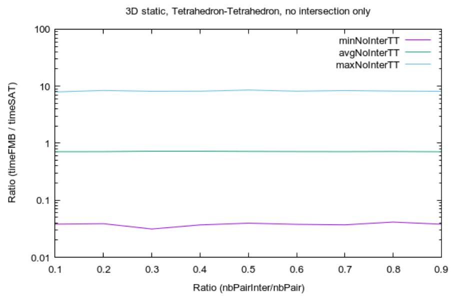








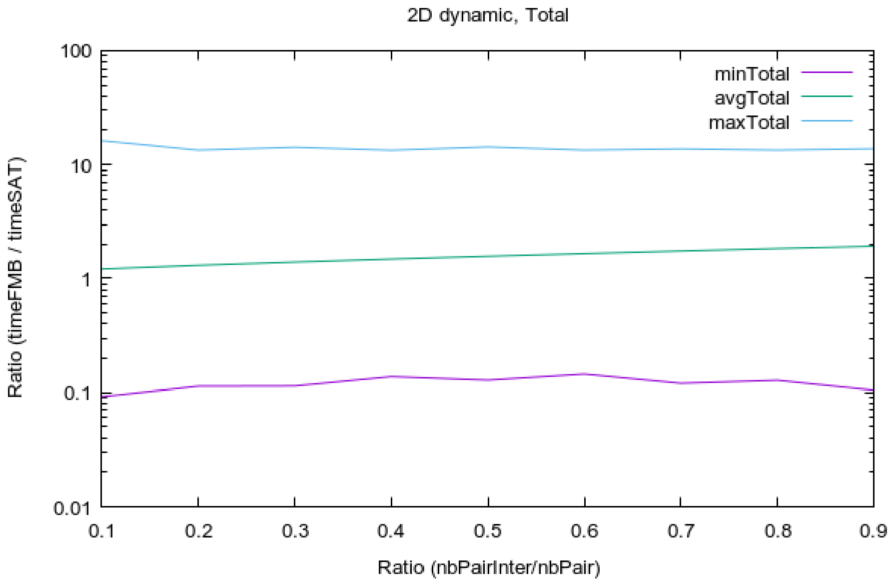


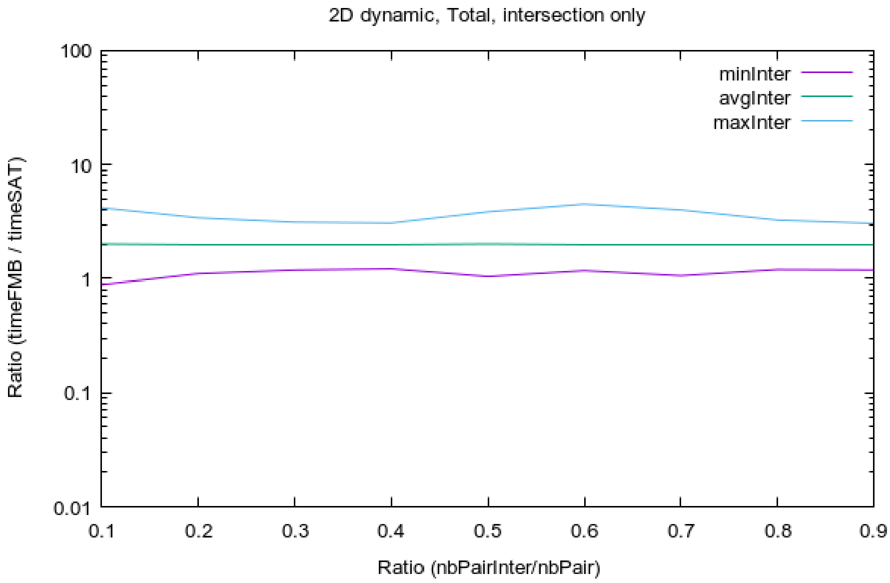


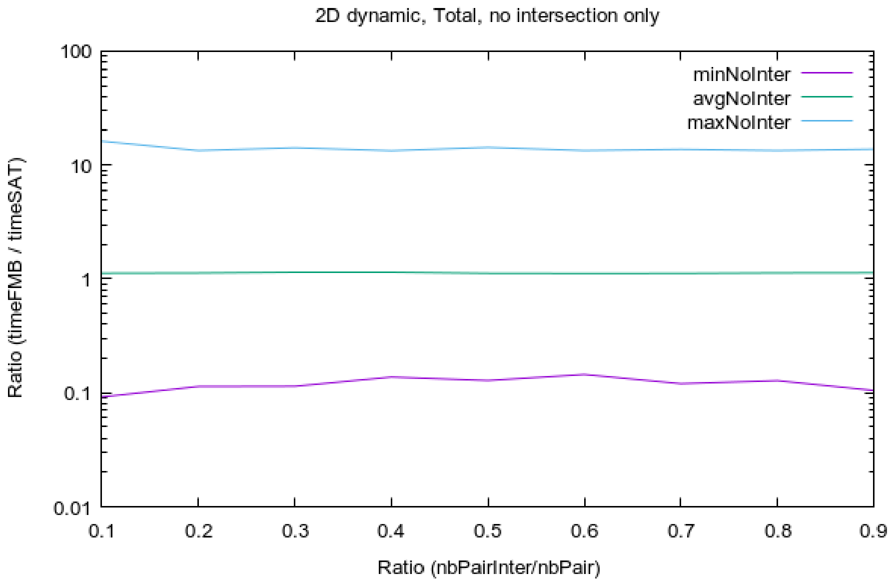
9.2.3 2D dynamic

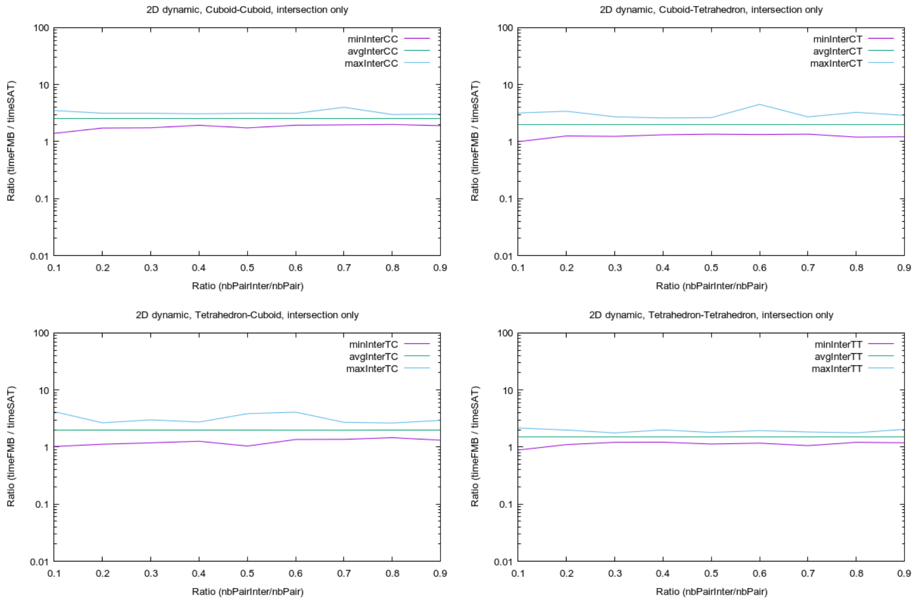
```
percPairInter   countInter      countNoInter    minInter        avgInter
        maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    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     74904   125088  0.887681        2.016953        4.174825
    0.091954        1.123632        16.230769       0.091954        1.212964
        16.230769       20132   29730   1.393305        2.538785
    3.503876        0.099291        1.081980        15.964286       0.099291
        1.227661        15.964286       18670   31428   0.996241
    1.974649        3.169118        0.103704        1.127723
    14.576923       0.103704        1.212415        14.576923       18712
    31296   1.023256        1.973773        4.174825        0.091954
    1.139832        16.230769       0.091954        1.223226
    16.230769       17390   32634   0.887681        1.504720        2.158537
        0.151786        1.142100        10.958333       0.151786
    1.178362        10.958333
0.2     74408   125588  1.104072        2.014288        3.407143
    0.113821        1.129257        13.416667       0.113821        1.306263
        13.416667       19804   29962   1.728643        2.538732
    3.121212        0.150943        1.060670        12.777778       0.150943
        1.356282        12.777778       18580   31448   1.253521
    1.975341        3.407143        0.113821        1.142465
    13.375000       0.113821        1.309040        13.375000       18536
    31290   1.121339        1.973529        2.659420        0.140187
    1.135709        13.416667       0.140187        1.303273
    13.416667       17488   32888   1.104072        1.504969        1.987342
        0.134454        1.172975        10.600000       0.134454
    1.239374        10.600000
0.3     74358   125638  1.186275        2.014763        3.122137
    0.114286        1.130617        14.208333       0.114286        1.395861
        14.208333       19750   30042   1.743719        2.538178
    3.122137        0.137615        1.061672        14.071429       0.137615
        1.504624        14.071429       18628   31770   1.233645
    1.974801        2.719745        0.114286        1.120817
    14.208333       0.114286        1.377012        14.208333       18738
    31306   1.186275        1.972418        3.000000        0.148148
    1.141991        13.720000       0.148148        1.391119
    13.720000       17242   32520   1.211340        1.504407        1.772152
        0.163265        1.192935        10.520000       0.163265
    1.286377        10.520000
0.4     74322   125672  1.216080        2.014599        3.068702
    0.137931        1.129479        13.375000       0.137931        1.483527
        13.375000       19900   30048   1.930636        2.537540
    3.068702        0.162963        1.111343        12.892857       0.162963
        1.681822        12.892857       18572   31356   1.315271
    1.973800        2.595588        0.145455        1.117667
    13.375000       0.145455        1.460120        13.375000       18456
    31598   1.264151        1.973124        2.743056        0.137931
    1.136260        13.291667       0.137931        1.471005
    13.291667       17394   32670   1.216080        1.503885        2.000000
        0.163265        1.150937        10.250000       0.163265
    1.292116        10.250000
```

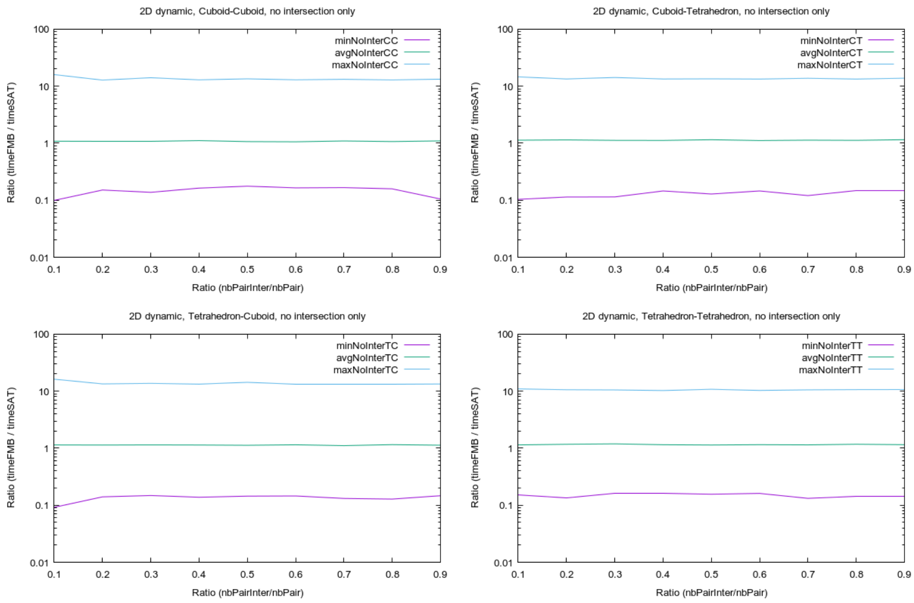
```
0.5     74472   125514  1.042636        2.018063        3.845070
    0.128571        1.120983        14.320000       0.128571        1.569523
            14.320000       20000   30204   1.736041        2.538760
    3.143939        0.176471        1.066312        13.464286       0.176471
            1.802536        13.464286       18764   31428   1.350515
    1.974423        2.633094        0.128571        1.151204
    13.458333       0.128571        1.562814        13.458333       18644
    30888   1.042636        1.973501        3.845070        0.144144
    1.125652        14.320000       0.144144        1.549576
    14.320000       17064   32994   1.129630        1.504453        1.801282
            0.155963        1.137871        10.800000       0.155963
    1.321162        10.800000
0.6     74282   125710  1.172589        2.014373        4.500000
    0.145455        1.118622        13.416667       0.145455        1.656072
            13.416667       19846   30348   1.936782        2.538218
    3.126866        0.164835        1.057982        12.928571       0.164835
            1.946124        12.928571       18562   31568   1.329949
    1.974937        4.500000        0.145455        1.112257
    13.333333       0.145455        1.629865        13.333333       18378
    31314   1.355330        1.974223        4.092199        0.145455
    1.148040        13.416667       0.145455        1.643750
    13.416667       17496   32480   1.172589        1.504179        1.943396
            0.160920        1.153105        10.307692       0.160920
    1.363749        10.307692
0.7     74126   125862  1.060606        2.015015        4.000000
    0.120567        1.120093        13.760000       0.120567        1.746539
            13.760000       19574   30328   1.965116        2.539467
    4.000000        0.166667        1.094054        13.178571       0.166667
            2.105843        13.178571       18724   31270   1.352041
    1.975064        2.715328        0.120567        1.129948
    13.760000       0.120567        1.721529        13.760000       18698
    31490   1.364103        1.973141        2.713287        0.131579
    1.109659        13.416667       0.131579        1.714097
    13.416667       17130   32774   1.060606        1.505115        1.838509
            0.132075        1.144809        10.565217       0.132075
    1.397023        10.565217
0.8     74850   125140  1.198198        2.014153        3.257353
    0.128205        1.130805        13.416667       0.128205        1.837484
            13.416667       19840   29836   2.000000        2.538889
    2.984848        0.158416        1.068731        12.851852       0.158416
            2.244857        12.851852       18640   31440   1.198198
    1.974329        3.257353        0.146789        1.121343
    13.333333       0.146789        1.803732        13.333333       18894
    31360   1.461538        1.973791        2.639706        0.128205
    1.153944        13.416667       0.128205        1.809821
    13.416667       17476   32504   1.210256        1.504551        1.779874
            0.144231        1.174611        10.652174       0.144231
    1.438563        10.652174
0.9     74332   125658  1.189320        2.014103        3.044776
    0.104972        1.133795        13.800000       0.104972        1.926072
            13.800000       19712   29930   1.897727        2.538934
    3.044776        0.104972        1.098386        13.259259       0.104972
            2.394879        13.259259       18660   31722   1.214286
    1.974944        2.884892        0.146789        1.153082
    13.800000       0.146789        1.892758        13.800000       18564
    31338   1.321543        1.973722        2.935714        0.146789
    1.128099        13.375000       0.146789        1.889160
    13.375000       17396   32668   1.189320        1.504494        2.043750
            0.144330        1.152971        10.695652       0.144330
    1.469342        10.695652
```


2D dynamic, Total
minTotal
avgTotal
maxTotal
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


2D dynamic, Total, intersection only
minInter
avgInter
maxInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)


2D dynamic, Total, no intersection only
minNoInter
avgNoInter
maxNoInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

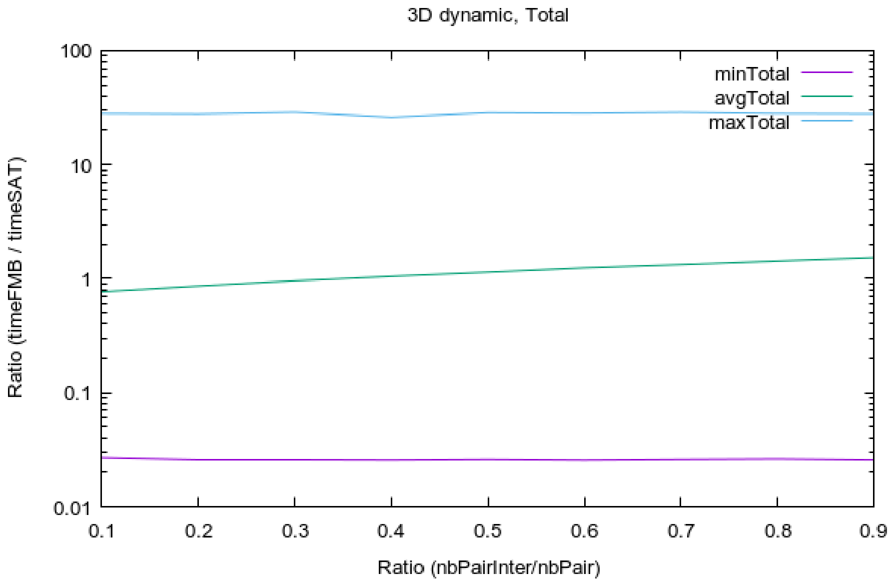


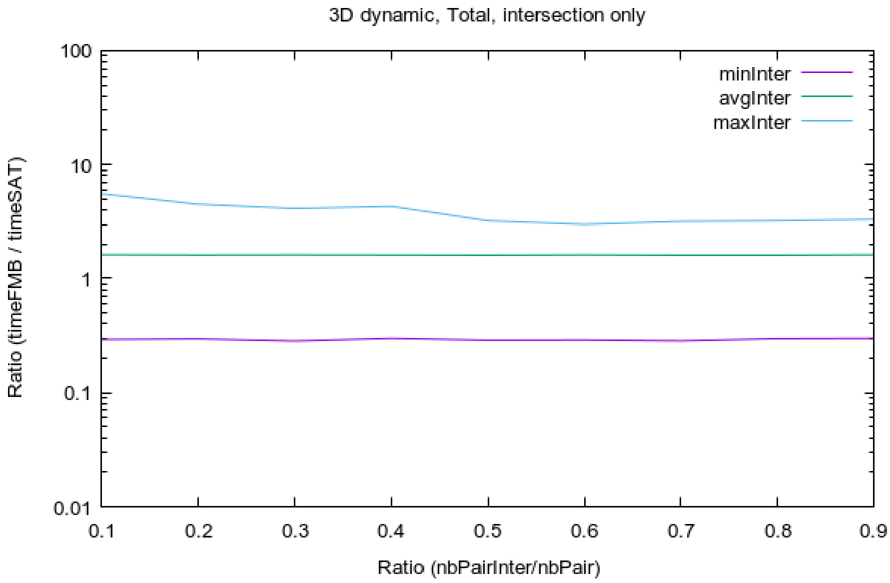


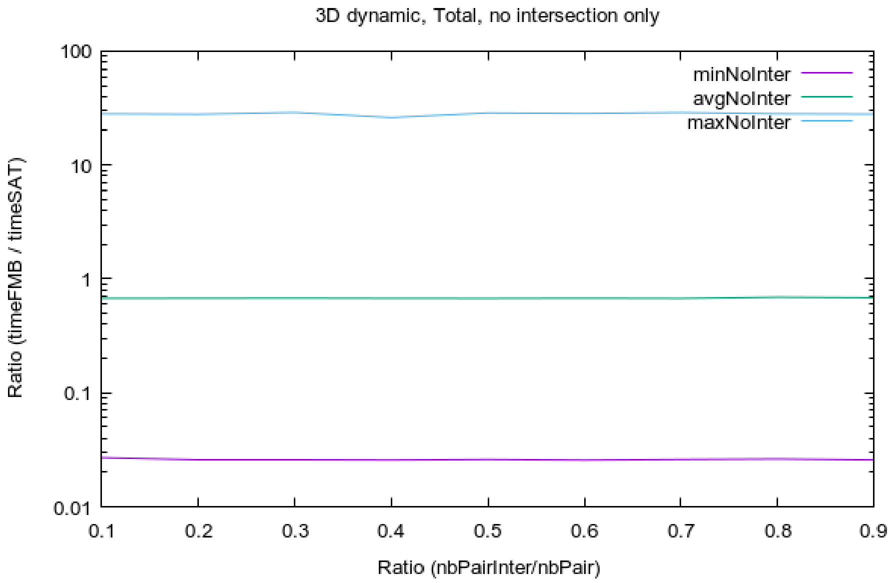
9.2.4 3D dynamic

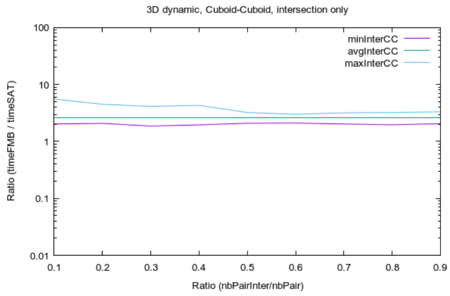
```
percPairInter   countInter      countNoInter    minInter        avgInter
        maxInter        minNoInter      avgNoInter      maxNoInter
    minTotal        avgTotal        maxTotal        countInterCC
    countNoInterCC  minInterCC      avgInterCC      maxInterCC
    minNoInterCC    avgNoInterCC    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     52834   147166  0.293536        1.621781        5.554522
    0.026941        0.673400        28.131579       0.026941        0.768238
        28.131579       16438   34000   2.030624        2.598016
    5.554522        0.037229        0.580840        12.801724       0.037229
        0.782557        12.801724       13292   36906   0.443189
    1.412200        2.560372        0.026941        0.670200
    21.884615       0.026941        0.744400        21.884615       13130
    36886   0.473267        1.412854        2.610092        0.027265
    0.675163        23.042553       0.027265        0.748932
    23.042553       9974    39374   0.293536        0.567197        1.144033
        0.027642        0.754674        28.131579       0.027642
    0.735926        28.131579
0.2     52496   147504  0.297245        1.611068        4.495770
    0.025680        0.673903        27.864865       0.025680        0.861336
        27.864865       16046   34134   2.079112        2.598359
    4.495770        0.035821        0.577536        12.952991       0.035821
        0.981700        12.952991       13314   36450   0.444610
    1.413174        3.315465        0.025895        0.678187
    21.760000       0.025895        0.825185        21.760000       12970
    36694   0.470246        1.410916        2.596774        0.026139
    0.667901        19.851852       0.026139        0.816504
    19.851852       10166   40226   0.297245        0.567261        0.927667
        0.025680        0.757268        27.864865       0.025680
    0.719267        27.864865
0.3     52580   147420  0.284794        1.615521        4.125326
    0.025699        0.676105        28.891892       0.025699        0.957930
        28.891892       16238   33772   1.860606        2.597153
    4.125326        0.037190        0.600904        12.893162       0.037190
        1.199778        12.893162       13096   36582   0.476882
    1.414129        3.504756        0.027195        0.658051
    20.377358       0.027195        0.884874        20.377358       13080
    36770   0.474621        1.413627        3.389014        0.026583
    0.669740        17.750000       0.026583        0.892906
    17.750000       10166   40296   0.284794        0.566778        0.851471
        0.025699        0.761331        28.891892       0.025699
    0.702965        28.891892
0.4     52136   147864  0.300667        1.610720        4.310680
    0.025641        0.673124        26.081081       0.025641        1.048163
        26.081081       15984   33904   1.953926        2.598292
    4.310680        0.036585        0.578696        12.817797       0.036585
        1.386535        12.817797       13032   36908   0.468912
    1.410582        2.571210        0.026415        0.675365
    22.183673       0.026415        0.969452        22.183673       12962
    36872   0.476412        1.412745        2.774218        0.025641
    0.672327        22.392157       0.025641        0.968494
    22.392157       10158   40180   0.300667        0.566126        1.253004
        0.025699        0.751476        26.081081       0.025699
    0.677336        26.081081
```

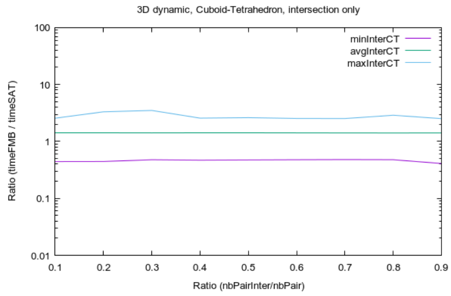
```
0.5     52316   147684  0.289695        1.603446        3.223881
    0.025954        0.672255        28.621622       0.025954        1.137850
        28.621622       15826   33988   2.090846        2.598208
    3.223881        0.038015        0.571280        12.838298       0.038015
        1.584744        12.838298       13010   36860   0.473443
    1.411593        2.620795        0.026194        0.680416
    21.346154       0.026194        1.046005        21.346154       13182
    36868   0.480808        1.409444        2.821536        0.026355
    0.659600        22.392157       0.026355        1.034522
    22.392157       10298   39968   0.289695        0.565401        0.768662
        0.025954        0.762271        28.621622       0.025954
    0.663836        28.621622
0.6     52058   147942  0.290957        1.616332        2.997412
    0.025622        0.675253        28.297297       0.025622        1.239901
        28.297297       16046   33684   2.104635        2.597714
    2.997412        0.037736        0.591699        13.037190       0.037736
        1.795308        13.037190       13154   37674   0.476882
    1.414123        2.533643        0.025777        0.663853
    21.196078       0.025777        1.114015        21.196078       12912
    36994   0.477554        1.410784        2.493426        0.025622
    0.675603        22.960000       0.025622        1.116712
    22.960000       9946    39590   0.290957        0.567332        0.765647
        0.025660        0.756865        28.297297       0.025660
    0.643145        28.297297
0.7     52652   147348  0.285806        1.606088        3.181495
    0.025974        0.671622        28.837838       0.025974        1.325748
        28.837838       16030   34170   2.026720        2.597686
    3.181495        0.037618        0.574623        13.247934       0.037618
        1.990767        13.247934       13290   36654   0.480078
    1.409727        2.526765        0.027174        0.675366
    20.865385       0.027174        1.189419        20.865385       13006
    36830   0.477044        1.410236        2.632716        0.026275
    0.662360        21.711538       0.026275        1.185873
    21.711538       10326   39694   0.285806        0.566149        0.772135
        0.025974        0.760259        28.837838       0.025974
    0.624382        28.837838
0.8     51706   148294  0.298597        1.607612        3.224184
    0.026255        0.689141        28.135135       0.026255        1.423917
        28.135135       15726   33722   1.962745        2.597139
    3.224184        0.037190        0.592550        12.886076       0.037190
        2.196221        12.886076       13024   37138   0.477513
    1.409518        2.887615        0.026255        0.689044
    22.153846       0.026255        1.265424        22.153846       12924
    37402   0.478232        1.411528        2.648587        0.026500
    0.683794        22.211538       0.026500        1.265981
    22.211538       10032   40032   0.298597        0.566228        0.829452
        0.026758        0.775593        28.135135       0.026758
    0.608101        28.135135
0.9     52292   147708  0.300539        1.619419        3.315975
    0.025738        0.681886        27.973684       0.025738        1.525666
        27.973684       16096   34110   2.048932        2.597994
    3.315975        0.036885        0.591668        13.038793       0.036885
        2.397361        13.038793       13338   36942   0.410185
    1.410209        2.519380        0.027135        0.680580
    22.211538       0.027135        1.337246        22.211538       13104
    36740   0.449684        1.414520        2.541764        0.026418
    0.680745        22.254902       0.026418        1.341143
    22.254902       9754    39916   0.300539        0.565933        0.752949
        0.025738        0.761239        27.973684       0.025738
    0.585464        27.973684
```

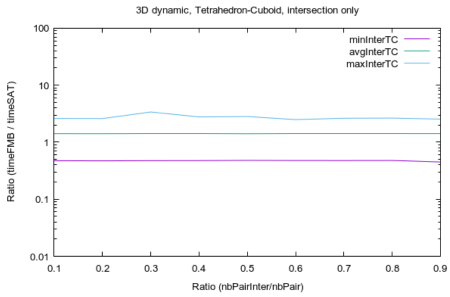

3D dynamic, Total
minTotal
avgTotal
maxTotal
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

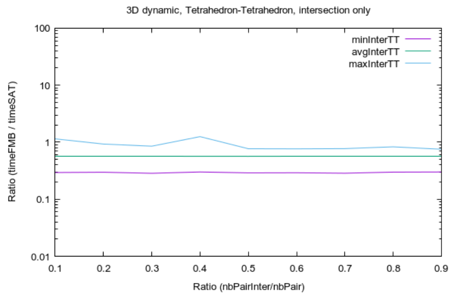

3D dynamic, Total, intersection only
minInter
avgInter
maxInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

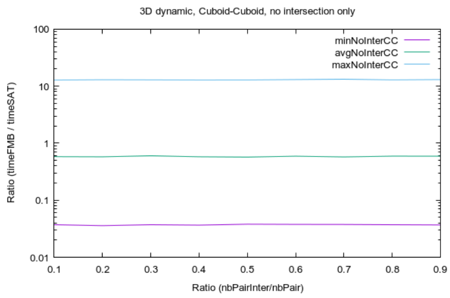

3D dynamic, Total, no intersection only
minNoInter
avgNoInter
maxNoInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (nbPairInter/nbPair)

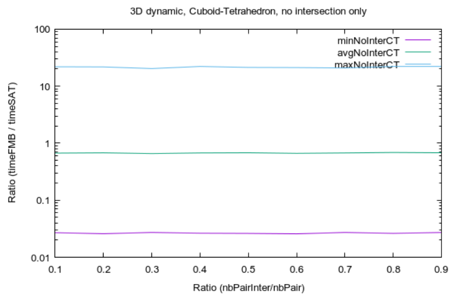

3D dynamic, Cuboid-Cuboid, intersection only
minInterCC
avgInterCC
maxInterCC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

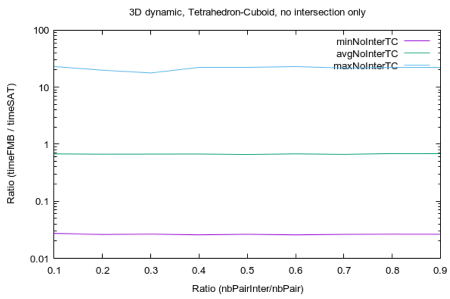

3D dynamic, Cuboid-Tetrahedron, intersection only
minInterCT
avgInterCT
maxInterCT
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)

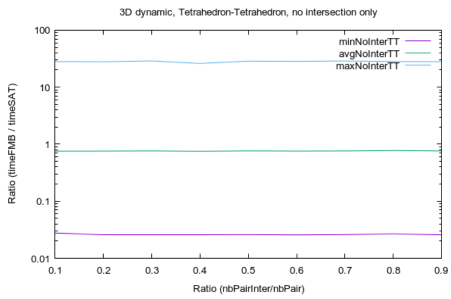

3D dynamic, Tetrahedron-Cuboid, intersection only
minInterTC
avgInterTC
maxInterTC
Ratio (timeFMB / timeSAT)
Ratio (nbPairInter/nbPair)


3D dynamic, Tetrahedron-Tetrahedron, intersection only
minInterTT
avgInterTT
maxInterTT
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)

10 Conclusion

The validation proves that the FMB algorithm correctly identifies intersection of pairs of Frames in accordance with the results of the SAT algorithm.

The qualification proves that the FMB algorithm is in average 50% slower than the SAT algorithm in 2D, and 17% faster in 3D.

11 Annex

11.1 SAT implementation

In this section I introduce the code of the implementation of the SAT algorithm, used to validate and qualify the FMB algorithm.

11.1.1 Header

```
#ifndef __SAT_H_
#define __SAT_H_

#include <stdbool.h>
#include <string.h>
#include "frame.h"

// ------------ Functions declaration -------------

// Test for intersection between 2D Frame 'that' and 2D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2D(
  const Frame2D* const that,
  const Frame2D* const tho);

// Test for intersection between 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
```

11.1.2 Body

```
#include "sat.h"

// ------------ Macros -------------

#define EPSILON 0.0000001

// ------------ Functions declaration -------------

// Check the intersection constraint along one axis
bool CheckAxis3D(
  const Frame3D* const that,
  const Frame3D* const tho,
  const double* const axis);

// Check the intersection constraint along one axis
bool CheckAxis3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho,
  const double* const axis,
  const double* const relSpeed);

// ------------ Functions implementation -------------

// Test for intersection between 2D Frame 'that' and 2D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2D(
  const Frame2D* const that,
  const Frame2D* const tho) {

  // Declare a variable to loop on Frames and commonalize code
  const Frame2D* frameEdge = that;

  // Loop to commonalize code when checking SAT based on that's edges
  // and then tho's edges
  for (int iFrame = 2;
       iFrame--;) {

    // Shortcuts
    FrameType frameEdgeType = frameEdge->type;
    const double* frameEdgeCompA = frameEdge->comp[0];
    const double* frameEdgeCompB = frameEdge->comp[1];

    // Declare a variable to memorize the number of edges, by default 2
    int nbEdges = 2;

    // Declare a variable to memorize the third edge in case of
    // tetrahedron
    double thirdEdge[2];

    // If the frame is a tetrahedron
    if (frameEdgeType == FrameTetrahedron) {

      // Initialise the third edge
      thirdEdge[0] = frameEdgeCompB[0] - frameEdgeCompA[0];
      thirdEdge[1] = frameEdgeCompB[1] - frameEdgeCompA[1];
```

```
    // Correct the number of edges
    nbEdges = 3;

  }

  // Loop on the frame's edges
  for (int iEdge = nbEdges;
       iEdge--;) {

    // Get the current edge
    const double* edge =
      (iEdge == 2 ? thirdEdge : frameEdge->comp[iEdge]);

    // Declare variables to memorize the boundaries of projection
    // of the two frames on the current edge
    double bdgBoxA[2];
    double bdgBoxB[2];

    // Declare two variables to loop on Frames and commonalize code
    const Frame2D* frame = that;
    double* bdgBox = bdgBoxA;

    // Loop on Frames
    for (int iFrame = 2;
         iFrame--;) {

      // Shortcuts
      const double* frameOrig = frame->orig;
      const double* frameCompA = frame->comp[0];
      const double* frameCompB = frame->comp[1];
      FrameType frameType = frame->type;

      // Get the number of vertices of frame
      int nbVertices = (frameType == FrameTetrahedron ? 3 : 4);

      // Declare a variable to memorize if the current vertex is
      // the first in the loop, used to initialize the boundaries
      bool firstVertex = true;

      // Loop on vertices of the frame
      for (int iVertex = nbVertices;
           iVertex--;) {

        // Get the vertex
        double vertex[2];
        vertex[0] = frameOrig[0];
        vertex[1] = frameOrig[1];
        switch (iVertex) {
          case 3:
            vertex[0] += frameCompA[0] + frameCompB[0];
            vertex[1] += frameCompA[1] + frameCompB[1];
            break;
          case 2:
            vertex[0] += frameCompA[0];
            vertex[1] += frameCompA[1];
            break;
          case 1:
            vertex[0] += frameCompB[0];
            vertex[1] += frameCompB[1];
            break;
          default:
```

```
            break;
          }

          // Get the projection of the vertex on the normal of the edge
          // Orientation of the normal doesn't matter, so we
          // use arbitrarily the normal (edge[1], -edge[0])
          double proj = vertex[0] * edge[1] - vertex[1] * edge[0];

          // If it's the first vertex
          if (firstVertex == true) {

              // Initialize the boundaries of the projection of the
              // Frame on the edge
              bdgBox[0] = proj;
              bdgBox[1] = proj;

              // Update the flag to memorize we did the first vertex
              firstVertex = false;

          // Else, it's not the first vertex
          } else {

            // Update the boundaries of the projection of the Frame on
            // the edge
            if (bdgBox[0] > proj)
              bdgBox[0] = proj;

            if (bdgBox[1] < proj)
              bdgBox[1] = proj;

          }

        }

        // Switch the frame to check the vertices of the second Frame
        frame = tho;
        bdgBox = bdgBoxB;

      }

      // If the projections of the two frames on the edge are
      // not intersecting
      if (bdgBoxB[1] < bdgBoxA[0] ||
          bdgBoxA[1] < bdgBoxB[0]) {

         // There exists an axis which separates the Frames,
         // thus they are not in intersection
         return false;

      }

    }

    // Switch the frames to test against the second Frame's edges
    frameEdge = tho;

  }

  // If we reaches here, it means the two Frames are intersecting
  return true;

}
```

```
// Test for intersection between moving 2D Frame 'that' and 2D
// Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2DTime(
  const Frame2DTime* const that,
  const Frame2DTime* const tho) {

  // Declare a variable to loop on Frames and commonalize code
  const Frame2DTime* frameEdge = that;

  // Declare a variable to memorize the speed of tho relative to that
  double relSpeed[2];
  relSpeed[0] = tho->speed[0] - that->speed[0];
  relSpeed[1] = tho->speed[1] - that->speed[1];

  // Loop to commonalize code when checking SAT based on that's edges
  // and then tho's edges
  for (int iFrame = 2;
       iFrame--;) {

    // Shortcuts
    FrameType frameEdgeType = frameEdge->type;
    const double* frameEdgeCompA = frameEdge->comp[0];
    const double* frameEdgeCompB = frameEdge->comp[1];

    // Declare a variable to memorize the number of edges, by default 2
    int nbEdges = 2;

    // Declare a variable to memorize the third edge in case of
    // tetrahedron
    double thirdEdge[2];

    // If the frame is a tetrahedron
    if (frameEdgeType == FrameTetrahedron) {

      // Initialise the third edge
      thirdEdge[0] = frameEdgeCompB[0] - frameEdgeCompA[0];
      thirdEdge[1] = frameEdgeCompB[1] - frameEdgeCompA[1];

      // Correct the number of edges
      nbEdges = 3;

    }

    // If the current frame is the second frame
    if (iFrame == 1) {

      // Add one more edge to take into account the movement
      // of tho relative to that
      ++nbEdges;

    }

    // Loop on the frame's edges
    for (int iEdge = nbEdges;
         iEdge--;) {

      // Get the current edge
      const double* edge =
        (iEdge == 3 ? relSpeed :
          (iEdge == 2 ?
```

```
      (frameEdgeType == FrameTetrahedron ? thirdEdge : relSpeed) :
      frameEdge->comp[iEdge]));

// Declare variables to memorize the boundaries of projection
// of the two frames on the current edge
double bdgBoxA[2];
double bdgBoxB[2];

// Declare two variables to loop on Frames and commonalize code
const Frame2DTime* frame = that;
double* bdgBox = bdgBoxA;

// Loop on Frames
for (int iFrame = 2;
     iFrame--;) {

  // Shortcuts
  const double* frameOrig = frame->orig;
  const double* frameCompA = frame->comp[0];
  const double* frameCompB = frame->comp[1];
  FrameType frameType = frame->type;

  // Get the number of vertices of frame
  int nbVertices = (frameType == FrameTetrahedron ? 3 : 4);

  // Declare a variable to memorize if the current vertex is
  // the first in the loop, used to initialize the boundaries
  bool firstVertex = true;

  // Loop on vertices of the frame
  for (int iVertex = nbVertices;
       iVertex--;) {

    // Get the vertex
    double vertex[2];
    vertex[0] = frameOrig[0];
    vertex[1] = frameOrig[1];
    switch (iVertex) {
      case 3:
        vertex[0] += frameCompA[0] + frameCompB[0];
        vertex[1] += frameCompA[1] + frameCompB[1];
        break;
      case 2:
        vertex[0] += frameCompA[0];
        vertex[1] += frameCompA[1];
        break;
      case 1:
        vertex[0] += frameCompB[0];
        vertex[1] += frameCompB[1];
        break;
      default:
        break;
    }

    // Get the projection of the vertex on the normal of the edge
    // Orientation of the normal doesn't matter, so we
    // use arbitrarily the normal (edge[1], -edge[0])
    double proj = vertex[0] * edge[1] - vertex[1] * edge[0];

    // If it's the first vertex
    if (firstVertex == true) {
```

```
          // Initialize the boundaries of the projection of the
          // Frame on the edge
          bdgBox[0] = proj;
          bdgBox[1] = proj;

          // Update the flag to memorize we did the first vertex
          firstVertex = false;

      // Else, it's not the first vertex
      } else {

        // Update the boundaries of the projection of the Frame on
        // the edge
        if (bdgBox[0] > proj)
          bdgBox[0] = proj;

        if (bdgBox[1] < proj)
          bdgBox[1] = proj;

      }

      // If we are checking the second frame's vertices
      if (frame == tho) {

        // Check also the vertices moved by the relative speed
        vertex[0] += relSpeed[0];
        vertex[1] += relSpeed[1];

        proj = vertex[0] * edge[1] - vertex[1] * edge[0];

        if (bdgBox[0] > proj)
          bdgBox[0] = proj;

        if (bdgBox[1] < proj)
          bdgBox[1] = proj;

      }

    }

    // Switch the frame to check the vertices of the second Frame
    frame = tho;
    bdgBox = bdgBoxB;

  }

  // If the projections of the two frames on the edge are
  // not intersecting
  if (bdgBoxB[1] < bdgBoxA[0] ||
      bdgBoxA[1] < bdgBoxB[0]) {

     // There exists an axis which separates the Frames,
     // thus they are not in intersection
     return false;

  }

}

// Switch the frames to test against the second Frame's edges
frameEdge = tho;
```

```
  }

  // If we reaches here, it means the two Frames are intersecting
  return true;

}

// Test for intersection between 3D Frame 'that' and 3D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3D(
  const Frame3D* const that,
  const Frame3D* const tho) {

  // Declare two variables to memorize the opposite edges in case
  // of tetrahedron
  double oppEdgesThat[3][3];
  double oppEdgesTho[3][3];

  // Declare two variables to memorize the number of edges, by default 3
  int nbEdgesThat = 3;
  int nbEdgesTho = 3;

  // If the first Frame is a tetrahedron
  if (that->type == FrameTetrahedron) {

    // Shortcuts
    const double* frameCompA = that->comp[0];
    const double* frameCompB = that->comp[1];
    const double* frameCompC = that->comp[2];

    // Initialise the opposite edges
    oppEdgesThat[0][0] = frameCompB[0] - frameCompA[0];
    oppEdgesThat[0][1] = frameCompB[1] - frameCompA[1];
    oppEdgesThat[0][2] = frameCompB[2] - frameCompA[2];

    oppEdgesThat[1][0] = frameCompB[0] - frameCompC[0];
    oppEdgesThat[1][1] = frameCompB[1] - frameCompC[1];
    oppEdgesThat[1][2] = frameCompB[2] - frameCompC[2];

    oppEdgesThat[2][0] = frameCompC[0] - frameCompA[0];
    oppEdgesThat[2][1] = frameCompC[1] - frameCompA[1];
    oppEdgesThat[2][2] = frameCompC[2] - frameCompA[2];

    // Correct the number of edges
    nbEdgesThat = 6;

  }

  // If the second Frame is a tetrahedron
  if (tho->type == FrameTetrahedron) {

    // Shortcuts
    const double* frameCompA = tho->comp[0];
    const double* frameCompB = tho->comp[1];
    const double* frameCompC = tho->comp[2];

    // Initialise the opposite edges
    oppEdgesTho[0][0] = frameCompB[0] - frameCompA[0];
    oppEdgesTho[0][1] = frameCompB[1] - frameCompA[1];
    oppEdgesTho[0][2] = frameCompB[2] - frameCompA[2];

    oppEdgesTho[1][0] = frameCompB[0] - frameCompC[0];
```

```
  oppEdgesTho[1][1] = frameCompB[1] - frameCompC[1];
  oppEdgesTho[1][2] = frameCompB[2] - frameCompC[2];

  oppEdgesTho[2][0] = frameCompC[0] - frameCompA[0];
  oppEdgesTho[2][1] = frameCompC[1] - frameCompA[1];
  oppEdgesTho[2][2] = frameCompC[2] - frameCompA[2];

  // Correct the number of edges
  nbEdgesTho = 6;

}

// Declare variables to loop on Frames and commonalize code
const Frame3D* frame = that;
const double (*oppEdgesA)[3] = oppEdgesThat;

// Loop to commonalize code when checking SAT based on that's edges
// and then tho's edges
for (int iFrame = 2;
     iFrame--;) {

  // Shortcuts
  FrameType frameType = frame->type;
  const double* frameCompA = frame->comp[0];
  const double* frameCompB = frame->comp[1];
  const double* frameCompC = frame->comp[2];

  // Declare a variable to memorize the number of faces, by default 3
  int nbFaces = 3;

  // Declare a variable to memorize the normal to faces
  // Arrangement is normFaces[iFace][iAxis]
  double normFaces[4][3];

  // Initialise the normal to faces
  normFaces[0][0] =
    frameCompA[1] * frameCompB[2] -
    frameCompA[2] * frameCompB[1];
  normFaces[0][1] =
    frameCompA[2] * frameCompB[0] -
    frameCompA[0] * frameCompB[2];
  normFaces[0][2] =
    frameCompA[0] * frameCompB[1] -
    frameCompA[1] * frameCompB[0];

  normFaces[1][0] =
    frameCompA[1] * frameCompC[2] -
    frameCompA[2] * frameCompC[1];
  normFaces[1][1] =
    frameCompA[2] * frameCompC[0] -
    frameCompA[0] * frameCompC[2];
  normFaces[1][2] =
    frameCompA[0] * frameCompC[1] -
    frameCompA[1] * frameCompC[0];

  normFaces[2][0] =
    frameCompC[1] * frameCompB[2] -
    frameCompC[2] * frameCompB[1];
  normFaces[2][1] =
    frameCompC[2] * frameCompB[0] -
    frameCompC[0] * frameCompB[2];
  normFaces[2][2] =
```

```
    frameCompC[0] * frameCompB[1] -
    frameCompC[1] * frameCompB[0];

  // If the frame is a tetrahedron
  if (frameType == FrameTetrahedron) {

    // Shortcuts
    const double* oppEdgeA = oppEdgesA[0];
    const double* oppEdgeB = oppEdgesA[1];

    // Initialise the normal to the opposite face
    normFaces[3][0] =
      oppEdgeA[1] * oppEdgeB[2] -
      oppEdgeA[2] * oppEdgeB[1];
    normFaces[3][1] =
      oppEdgeA[2] * oppEdgeB[0] -
      oppEdgeA[0] * oppEdgeB[2];
    normFaces[3][2] =
      oppEdgeA[0] * oppEdgeB[1] -
      oppEdgeA[1] * oppEdgeB[0];

    // Correct the number of faces
    nbFaces = 4;

  }

  // Loop on the frame's faces
  for (int iFace = nbFaces;
       iFace--;) {

    // Check against the current face's normal
    bool isIntersection =
      CheckAxis3D(
        that,
        tho,
        normFaces[iFace]);

    // If the axis is separating the Frames
    if (isIntersection == false) {

      // The Frames are not in intersection,
      // terminate the test
      return false;

    }

  }

  // Switch the frame to test against the second Frame
  frame = tho;
  oppEdgesA = oppEdgesTho;

}

// Loop on the pair of edges between the two frames
for (int iEdgeThat = nbEdgesThat;
     iEdgeThat--;) {

  // Get the first edge
  const double* edgeThat =
    (iEdgeThat < 3 ?
      that->comp[iEdgeThat] :
```

```
        oppEdgesThat[iEdgeThat - 3]);

    for (int iEdgeTho = nbEdgesTho;
         iEdgeTho--;) {

      // Get the second edge
      const double* edgeTho =
        (iEdgeTho < 3 ?
          tho->comp[iEdgeTho] :
          oppEdgesTho[iEdgeTho - 3]);

      // Get the cross product of the two edges
      double axis[3];
      axis[0] = edgeThat[1] * edgeTho[2] - edgeThat[2] * edgeTho[1];
      axis[1] = edgeThat[2] * edgeTho[0] - edgeThat[0] * edgeTho[2];
      axis[2] = edgeThat[0] * edgeTho[1] - edgeThat[1] * edgeTho[0];

      // Check against the cross product of the two edges
      bool isIntersection =
        CheckAxis3D(
          that,
          tho,
          axis);

      // If the axis is separating the Frames
      if (isIntersection == false) {

        // The Frames are not in intersection,
        // terminate the test
        return false;

      }

    }

  }

  // If we reaches here, it means the two Frames are intersecting
  return true;

}

// Test for intersection between moving 3D Frame 'that' and 3D
// Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho) {

  // Declare two variables to memorize the opposite edges in case
  // of tetrahedron
  double oppEdgesThat[3][3];
  double oppEdgesTho[3][3];

  // Declare a variable to memorize the speed of tho relative to that
  double relSpeed[3];
  relSpeed[0] = tho->speed[0] - that->speed[0];
  relSpeed[1] = tho->speed[1] - that->speed[1];
  relSpeed[2] = tho->speed[2] - that->speed[2];

  // Declare two variables to memorize the number of edges, by default 3
  int nbEdgesThat = 3;
```

```
int nbEdgesTho = 3;

// If the first Frame is a tetrahedron
if (that->type == FrameTetrahedron) {

  // Shortcuts
  const double* frameCompA = that->comp[0];
  const double* frameCompB = that->comp[1];
  const double* frameCompC = that->comp[2];

  // Initialise the opposite edges
  oppEdgesThat[0][0] = frameCompB[0] - frameCompA[0];
  oppEdgesThat[0][1] = frameCompB[1] - frameCompA[1];
  oppEdgesThat[0][2] = frameCompB[2] - frameCompA[2];

  oppEdgesThat[1][0] = frameCompB[0] - frameCompC[0];
  oppEdgesThat[1][1] = frameCompB[1] - frameCompC[1];
  oppEdgesThat[1][2] = frameCompB[2] - frameCompC[2];

  oppEdgesThat[2][0] = frameCompC[0] - frameCompA[0];
  oppEdgesThat[2][1] = frameCompC[1] - frameCompA[1];
  oppEdgesThat[2][2] = frameCompC[2] - frameCompA[2];

  // Correct the number of edges
  nbEdgesThat = 6;

}

// If the second Frame is a tetrahedron
if (tho->type == FrameTetrahedron) {

  // Shortcuts
  const double* frameCompA = tho->comp[0];
  const double* frameCompB = tho->comp[1];
  const double* frameCompC = tho->comp[2];

  // Initialise the opposite edges
  oppEdgesTho[0][0] = frameCompB[0] - frameCompA[0];
  oppEdgesTho[0][1] = frameCompB[1] - frameCompA[1];
  oppEdgesTho[0][2] = frameCompB[2] - frameCompA[2];

  oppEdgesTho[1][0] = frameCompB[0] - frameCompC[0];
  oppEdgesTho[1][1] = frameCompB[1] - frameCompC[1];
  oppEdgesTho[1][2] = frameCompB[2] - frameCompC[2];

  oppEdgesTho[2][0] = frameCompC[0] - frameCompA[0];
  oppEdgesTho[2][1] = frameCompC[1] - frameCompA[1];
  oppEdgesTho[2][2] = frameCompC[2] - frameCompA[2];

  // Correct the number of edges
  nbEdgesTho = 6;

}

// Declare variables to loop on Frames and commonalize code
const Frame3DTime* frame = that;
const double (*oppEdgesA)[3] = oppEdgesThat;

// Loop to commonalize code when checking SAT based on that's edges
// and then tho's edges
for (int iFrame = 2;
     iFrame--;) {
```

```
// Shortcuts
FrameType frameType = frame->type;
const double* frameCompA = frame->comp[0];
const double* frameCompB = frame->comp[1];
const double* frameCompC = frame->comp[2];

// Declare a variable to memorize the number of faces, by default 3
int nbFaces = 3;

// Declare a variable to memorize the normal to faces
// Arrangement is normFaces[iFace][iAxis]
double normFaces[10][3];

// Initialise the normal to faces
normFaces[0][0] =
  frameCompA[1] * frameCompB[2] -
  frameCompA[2] * frameCompB[1];
normFaces[0][1] =
  frameCompA[2] * frameCompB[0] -
  frameCompA[0] * frameCompB[2];
normFaces[0][2] =
  frameCompA[0] * frameCompB[1] -
  frameCompA[1] * frameCompB[0];

normFaces[1][0] =
  frameCompA[1] * frameCompC[2] -
  frameCompA[2] * frameCompC[1];
normFaces[1][1] =
  frameCompA[2] * frameCompC[0] -
  frameCompA[0] * frameCompC[2];
normFaces[1][2] =
  frameCompA[0] * frameCompC[1] -
  frameCompA[1] * frameCompC[0];

normFaces[2][0] =
  frameCompC[1] * frameCompB[2] -
  frameCompC[2] * frameCompB[1];
normFaces[2][1] =
  frameCompC[2] * frameCompB[0] -
  frameCompC[0] * frameCompB[2];
normFaces[2][2] =
  frameCompC[0] * frameCompB[1] -
  frameCompC[1] * frameCompB[0];

// If the frame is a tetrahedron
if (frameType == FrameTetrahedron) {

  // Shortcuts
  const double* oppEdgeA = oppEdgesA[0];
  const double* oppEdgeB = oppEdgesA[1];

  // Initialise the normal to the opposite face
  normFaces[3][0] =
    oppEdgeA[1] * oppEdgeB[2] -
    oppEdgeA[2] * oppEdgeB[1];
  normFaces[3][1] =
    oppEdgeA[2] * oppEdgeB[0] -
    oppEdgeA[0] * oppEdgeB[2];
  normFaces[3][2] =
    oppEdgeA[0] * oppEdgeB[1] -
    oppEdgeA[1] * oppEdgeB[0];
```

```
  // Correct the number of faces
  nbFaces = 4;

}

// If we are checking the frame 'tho'
if (frame == tho) {

  // Add the normal to the virtual faces created by the speed
  // of tho relative to that

  normFaces[nbFaces][0] =
    relSpeed[1] * frameCompA[2] -
    relSpeed[2] * frameCompA[1];
  normFaces[nbFaces][1] =
    relSpeed[2] * frameCompA[0] -
    relSpeed[0] * frameCompA[2];
  normFaces[nbFaces][2] =
    relSpeed[0] * frameCompA[1] -
    relSpeed[1] * frameCompA[0];
  if (fabs(normFaces[nbFaces][0]) > EPSILON ||
      fabs(normFaces[nbFaces][1]) > EPSILON ||
      fabs(normFaces[nbFaces][2]) > EPSILON)
    ++nbFaces;

  normFaces[nbFaces][0] =
    relSpeed[1] * frameCompB[2] -
    relSpeed[2] * frameCompB[1];
  normFaces[nbFaces][1] =
    relSpeed[2] * frameCompB[0] -
    relSpeed[0] * frameCompB[2];
  normFaces[nbFaces][2] =
    relSpeed[0] * frameCompB[1] -
    relSpeed[1] * frameCompB[0];
  if (fabs(normFaces[nbFaces][0]) > EPSILON ||
      fabs(normFaces[nbFaces][1]) > EPSILON ||
      fabs(normFaces[nbFaces][2]) > EPSILON)
    ++nbFaces;

  normFaces[nbFaces][0] =
    relSpeed[1] * frameCompC[2] -
    relSpeed[2] * frameCompC[1];
  normFaces[nbFaces][1] =
    relSpeed[2] * frameCompC[0] -
    relSpeed[0] * frameCompC[2];
  normFaces[nbFaces][2] =
    relSpeed[0] * frameCompC[1] -
    relSpeed[1] * frameCompC[0];
  if (fabs(normFaces[nbFaces][0]) > EPSILON ||
      fabs(normFaces[nbFaces][1]) > EPSILON ||
      fabs(normFaces[nbFaces][2]) > EPSILON)
    ++nbFaces;

  if (frameType == FrameTetrahedron) {

    const double* oppEdgeA = oppEdgesA[0];
    const double* oppEdgeB = oppEdgesA[1];
    const double* oppEdgeC = oppEdgesA[2];

    normFaces[nbFaces][0] =
      relSpeed[1] * oppEdgeA[2] -
```

```
      relSpeed[2] * oppEdgeA[1];
    normFaces[nbFaces][1] =
      relSpeed[2] * oppEdgeA[0] -
      relSpeed[0] * oppEdgeA[2];
    normFaces[nbFaces][2] =
      relSpeed[0] * oppEdgeA[1] -
      relSpeed[1] * oppEdgeA[0];
    if (fabs(normFaces[nbFaces][0]) > EPSILON ||
        fabs(normFaces[nbFaces][1]) > EPSILON ||
        fabs(normFaces[nbFaces][2]) > EPSILON)
      ++nbFaces;

    normFaces[nbFaces][0] =
      relSpeed[1] * oppEdgeB[2] -
      relSpeed[2] * oppEdgeB[1];
    normFaces[nbFaces][1] =
      relSpeed[2] * oppEdgeB[0] -
      relSpeed[0] * oppEdgeB[2];
    normFaces[nbFaces][2] =
      relSpeed[0] * oppEdgeB[1] -
      relSpeed[1] * oppEdgeB[0];
    if (fabs(normFaces[nbFaces][0]) > EPSILON ||
        fabs(normFaces[nbFaces][1]) > EPSILON ||
        fabs(normFaces[nbFaces][2]) > EPSILON)
      ++nbFaces;

    normFaces[nbFaces][0] =
      relSpeed[1] * oppEdgeC[2] -
      relSpeed[2] * oppEdgeC[1];
    normFaces[nbFaces][1] =
      relSpeed[2] * oppEdgeC[0] -
      relSpeed[0] * oppEdgeC[2];
    normFaces[nbFaces][2] =
      relSpeed[0] * oppEdgeC[1] -
      relSpeed[1] * oppEdgeC[0];
    if (fabs(normFaces[nbFaces][0]) > EPSILON ||
        fabs(normFaces[nbFaces][1]) > EPSILON ||
        fabs(normFaces[nbFaces][2]) > EPSILON)
      ++nbFaces;

  }
}

// Loop on the frame's faces
for (int iFace = nbFaces;
     iFace--;) {

  // Check against the current face's normal
  bool isIntersection =
    CheckAxis3DTime(
      that,
      tho,
      normFaces[iFace],
      relSpeed);

  // If the axis is separating the Frames
  if (isIntersection == false) {

    // The Frames are not in intersection,
    // terminate the test
    return false;
```

```
      }

    }

    // Switch the frame to test against the second Frame
    frame = tho;
    oppEdgesA = oppEdgesTho;

  }

  // Loop on the pair of edges between the two frames
  for (int iEdgeThat = nbEdgesThat;
       iEdgeThat--;) {

    // Get the first edge
    const double* edgeThat =
      (iEdgeThat < 3 ?
        that->comp[iEdgeThat] :
        oppEdgesThat[iEdgeThat - 3]);

    for (int iEdgeTho = nbEdgesTho + 1;
         iEdgeTho--;) {

      // Get the second edge
      const double* edgeTho =
        (iEdgeTho == nbEdgesTho ?
          relSpeed :
          (iEdgeTho < 3 ?
            tho->comp[iEdgeTho] :
            oppEdgesTho[iEdgeTho - 3]));

      // Get the cross product of the two edges
      double axis[3];
      axis[0] = edgeThat[1] * edgeTho[2] - edgeThat[2] * edgeTho[1];
      axis[1] = edgeThat[2] * edgeTho[0] - edgeThat[0] * edgeTho[2];
      axis[2] = edgeThat[0] * edgeTho[1] - edgeThat[1] * edgeTho[0];

      // Check against the cross product of the two edges
      bool isIntersection =
        CheckAxis3DTime(
          that,
          tho,
          axis,
          relSpeed);

      // If the axis is separating the Frames
      if (isIntersection == false) {

        // The Frames are not in intersection,
        // terminate the test
        return false;

      }

    }

  }

  // If we reaches here, it means the two Frames are intersecting
  return true;

}
```

```
// Check the intersection constraint for Frames 'that' and 'tho'
// relatively to 'axis'
bool CheckAxis3D(
  const Frame3D* const that,
  const Frame3D* const tho,
  const double* const axis) {

  // Declare variables to memorize the boundaries of projection
  // of the two frames on the current edge
  double bdgBoxA[2];
  double bdgBoxB[2];

  // Declare two variables to loop on Frames and commonalize code
  const Frame3D* frame = that;
  double* bdgBox = bdgBoxA;

  // Loop on Frames
  for (int iFrame = 2;
       iFrame--;) {

    // Shortcuts
    const double* frameOrig = frame->orig;
    const double* frameCompA = frame->comp[0];
    const double* frameCompB = frame->comp[1];
    const double* frameCompC = frame->comp[2];
    FrameType frameType = frame->type;

    // Get the number of vertices of frame
    int nbVertices = (frameType == FrameTetrahedron ? 4 : 8);

    // Declare a variable to memorize if the current vertex is
    // the first in the loop, used to initialize the boundaries
    bool firstVertex = true;

    // Loop on vertices of the frame
    for (int iVertex = nbVertices;
         iVertex--;) {

      // Get the vertex
      double vertex[3];
      vertex[0] = frameOrig[0];
      vertex[1] = frameOrig[1];
      vertex[2] = frameOrig[2];
      switch (iVertex) {
        case 7:
          vertex[0] +=
            frameCompA[0] +  frameCompB[0] + frameCompC[0];
          vertex[1] +=
            frameCompA[1] +  frameCompB[1] + frameCompC[1];
          vertex[2] +=
            frameCompA[2] +  frameCompB[2] + frameCompC[2];
          break;
        case 6:
          vertex[0] += frameCompB[0] + frameCompC[0];
          vertex[1] += frameCompB[1] + frameCompC[1];
          vertex[2] += frameCompB[2] + frameCompC[2];
          break;
        case 5:
          vertex[0] += frameCompA[0] + frameCompC[0];
          vertex[1] += frameCompA[1] + frameCompC[1];
          vertex[2] += frameCompA[2] + frameCompC[2];
```

```
        break;
      case 4:
        vertex[0] += frameCompA[0] + frameCompB[0];
        vertex[1] += frameCompA[1] + frameCompB[1];
        vertex[2] += frameCompA[2] + frameCompB[2];
        break;
      case 3:
        vertex[0] += frameCompC[0];
        vertex[1] += frameCompC[1];
        vertex[2] += frameCompC[2];
        break;
      case 2:
        vertex[0] += frameCompB[0];
        vertex[1] += frameCompB[1];
        vertex[2] += frameCompB[2];
        break;
      case 1:
        vertex[0] += frameCompA[0];
        vertex[1] += frameCompA[1];
        vertex[2] += frameCompA[2];
        break;
      default:
        break;
    }

    // Get the projection of the vertex on the axis
    double proj =
      vertex[0] * axis[0] +
      vertex[1] * axis[1] +
      vertex[2] * axis[2];

    // If it's the first vertex
    if (firstVertex == true) {

        // Initialize the boundaries of the projection of the
        // Frame on the edge
        bdgBox[0] = proj;
        bdgBox[1] = proj;

        // Update the flag to memorize we did the first vertex
        firstVertex = false;

    // Else, it's not the first vertex
    } else {

      // Update the boundaries of the projection of the Frame on
      // the edge
      if (bdgBox[0] > proj)
        bdgBox[0] = proj;

      if (bdgBox[1] < proj)
        bdgBox[1] = proj;

    }

  }

  // Switch the frame to check the vertices of the second Frame
  frame = tho;
  bdgBox = bdgBoxB;

}
```

```
  // If the projections of the two frames on the edge are
  // not intersecting
  if (bdgBoxB[1] < bdgBoxA[0] ||
      bdgBoxA[1] < bdgBoxB[0]) {

     // There exists an axis which separates the Frames,
     // thus they are not in intersection
     return false;

  }

  // If we reaches here the two Frames are in intersection
  return true;

}

// Check the intersection constraint for Frames 'that' and 'tho'
// relatively to 'axis'
bool CheckAxis3DTime(
  const Frame3DTime* const that,
  const Frame3DTime* const tho,
  const double* const axis,
  const double* const relSpeed) {

  // Declare variables to memorize the boundaries of projection
  // of the two frames on the current edge
  double bdgBoxA[2];
  double bdgBoxB[2];

  // Declare two variables to loop on Frames and commonalize code
  const Frame3DTime* frame = that;
  double* bdgBox = bdgBoxA;

  // Loop on Frames
  for (int iFrame = 2;
       iFrame--;) {

    // Shortcuts
    const double* frameOrig = frame->orig;
    const double* frameCompA = frame->comp[0];
    const double* frameCompB = frame->comp[1];
    const double* frameCompC = frame->comp[2];
    FrameType frameType = frame->type;

    // Get the number of vertices of frame
    int nbVertices = (frameType == FrameTetrahedron ? 4 : 8);

    // Declare a variable to memorize if the current vertex is
    // the first in the loop, used to initialize the boundaries
    bool firstVertex = true;

    // Loop on vertices of the frame
    for (int iVertex = nbVertices;
         iVertex--;) {

      // Get the vertex
      double vertex[3];
      vertex[0] = frameOrig[0];
      vertex[1] = frameOrig[1];
      vertex[2] = frameOrig[2];
      switch (iVertex) {
```

```
  case 7:
    vertex[0] +=
      frameCompA[0] +  frameCompB[0] + frameCompC[0];
    vertex[1] +=
      frameCompA[1] +  frameCompB[1] + frameCompC[1];
    vertex[2] +=
      frameCompA[2] +  frameCompB[2] + frameCompC[2];
    break;
  case 6:
    vertex[0] += frameCompB[0] + frameCompC[0];
    vertex[1] += frameCompB[1] + frameCompC[1];
    vertex[2] += frameCompB[2] + frameCompC[2];
    break;
  case 5:
    vertex[0] += frameCompA[0] + frameCompC[0];
    vertex[1] += frameCompA[1] + frameCompC[1];
    vertex[2] += frameCompA[2] + frameCompC[2];
    break;
  case 4:
    vertex[0] += frameCompA[0] + frameCompB[0];
    vertex[1] += frameCompA[1] + frameCompB[1];
    vertex[2] += frameCompA[2] + frameCompB[2];
    break;
  case 3:
    vertex[0] += frameCompC[0];
    vertex[1] += frameCompC[1];
    vertex[2] += frameCompC[2];
    break;
  case 2:
    vertex[0] += frameCompB[0];
    vertex[1] += frameCompB[1];
    vertex[2] += frameCompB[2];
    break;
  case 1:
    vertex[0] += frameCompA[0];
    vertex[1] += frameCompA[1];
    vertex[2] += frameCompA[2];
    break;
  default:
    break;
}

// Get the projection of the vertex on the axis
double proj =
  vertex[0] * axis[0] +
  vertex[1] * axis[1] +
  vertex[2] * axis[2];

// If it's the first vertex
if (firstVertex == true) {

    // Initialize the boundaries of the projection of the
    // Frame on the edge
    bdgBox[0] = proj;
    bdgBox[1] = proj;

    // Update the flag to memorize we did the first vertex
    firstVertex = false;

// Else, it's not the first vertex
} else {
```

```
        // Update the boundaries of the projection of the Frame on
        // the edge
        if (bdgBox[0] > proj)
          bdgBox[0] = proj;

        if (bdgBox[1] < proj)
          bdgBox[1] = proj;

      }

      // If we are checking the second frame's vertices
      if (frame == tho) {

        // Check also the vertices moved by the relative speed
        vertex[0] += relSpeed[0];
        vertex[1] += relSpeed[1];
        vertex[2] += relSpeed[2];

      proj =
        vertex[0] * axis[0] +
        vertex[1] * axis[1] +
        vertex[2] * axis[2];

        if (bdgBox[0] > proj)
          bdgBox[0] = proj;

        if (bdgBox[1] < proj)
          bdgBox[1] = proj;

      }

    }

    // Switch the frame to check the vertices of the second Frame
    frame = tho;
    bdgBox = bdgBoxB;

  }

  // If the projections of the two frames on the edge are
  // not intersecting
  if (bdgBoxB[1] < bdgBoxA[0] ||
      bdgBoxA[1] < bdgBoxB[0]) {

     // There exists an axis which separates the Frames,
     // thus they are not in intersection
     return false;

  }

  // If we reaches here the two Frames are in intersection
  return true;

}
```

11.2 Makefile

In this section I introduce the Makefile used to compile the code given in the previous sections.

```
COMPILER=gcc
OPTIMIZATION=-O3

all : compile run plot doc

install :
	sudo apt-get install gnuplot

compile : main unitTests validation qualification

main : main2D main2DTime main3D main3DTime

main2D:
	cd 2D; make main; cd -

main2DTime:
	cd 2DTime; make main; cd -

main3D:
	cd 3D; make main; cd -

main3DTime:
	cd 3DTime; make main; cd -

unitTests : unitTests2D unitTests2DTime unitTests3D unitTests3DTime

unitTests2D:
	cd 2D; make unitTests; cd -

unitTests2DTime:
	cd 2DTime; make unitTests; cd -

unitTests3D:
	cd 3D; make unitTests; cd -

unitTests3DTime:
	cd 3DTime; make unitTests; cd -

validation : validation2D validation2DTime validation3D validation3DTime

validation2D:
	cd 2D; make validation; cd -

validation2DTime:
	cd 2DTime; make validation; cd -

validation3D:
	cd 3D; make validation; cd -

validation3DTime:
	cd 3DTime; make validation; cd -

qualification : qualification2D qualification2DTime qualification3D
    qualification3DTime

qualification2D:
	cd 2D; make qualification; cd -

qualification2DTime:
	cd 2DTime; make qualification; cd -

qualification3D:
```

```
	cd 3D; make qualification; cd -

qualification3DTime:
	cd 3DTime; make qualification; cd -

clean : clean2D clean2DTime clean3D clean3DTime

clean2D:
	cd 2D; make clean; cd -

clean2DTime:
	cd 2DTime; make clean; cd -

clean3D:
	cd 3D; make clean; cd -

clean3DTime:
	cd 3DTime; make clean; cd -

valgrind : valgrind2D valgrind2DTime valgrind3D valgrind3DTime

valgrind2D:
	cd 2D; make valgrind; cd -

valgrind2DTime:
	cd 2DTime; make valgrind; cd -

valgrind3D:
	cd 3D; make valgrind; cd -

valgrind3DTime:
	cd 3DTime; make valgrind; cd -

run : run2D run2DTime run3D run3DTime

run2D:
	cd 2D; ./main > ../Results/main2D.txt; ./unitTests > ../Results/
	    unitTests2D.txt; ./validation > ../Results/validation2D.txt;
	    grep failed ../Results/validation2D.txt; ./qualification > ../
	    Results/qualification2D.txt; grep failed ../Results/
	    qualification2D.txt; cd -

run3D:
	cd 3D; ./main > ../Results/main3D.txt; ./unitTests > ../Results/
	    unitTests3D.txt; ./validation > ../Results/validation3D.txt;
	    grep failed ../Results/validation3D.txt; ./qualification > ../
	    Results/qualification3D.txt; grep failed ../Results/
	    qualification3D.txt; cd -

run2DTime:
	cd 2DTime; ./main > ../Results/main2DTime.txt; ./unitTests > ../
	    Results/unitTests2DTime.txt; ./validation > ../Results/
	    validation2DTime.txt; grep failed ../Results/validation2DTime.
	    txt; ./qualification > ../Results/qualification2DTime.txt; grep
	    failed ../Results/qualification2DTime.txt; cd -

run3DTime:
	cd 3DTime; ./main > ../Results/main3DTime.txt; ./unitTests > ../
	    Results/unitTests3DTime.txt; ./validation > ../Results/
	    validation3DTime.txt; grep failed ../Results/validation3DTime.
	    txt; ./qualification > ../Results/qualification3DTime.txt; grep
	    failed ../Results/qualification3DTime.txt; cd -
```

```
plot: cleanPlot plot2D plot2DTime plot3D plot3DTime

cleanPlot:
	rm Results/*.png

plot2D:
	cd Results; gnuplot qualification2D.gnu < qualification2D.txt; cd -

plot2DTime:
	cd Results; gnuplot qualification2DTime.gnu < qualification2DTime.
	    txt; cd -

plot3D:
	cd Results; gnuplot qualification3D.gnu < qualification3D.txt; cd -

plot3DTime:
	cd Results; gnuplot qualification3DTime.gnu < qualification3DTime.
	    txt; cd -

doc:
	cd Doc; make latex; cd -
```

11.2.1 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
```

11.2.2 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
```

11.2.3 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
```

11.2.4 3D dynamic

```
all : main unitTests validation qualification

COMPILER?=gcc
OPTIMIZATION?=-O3
BUILD_ARG=$(OPTIMIZATION) -I../SAT -I../Frame
```

```
main : main.o fmb3dt.o frame.o Makefile
	$(COMPILER) -o main main.o fmb3dt.o frame.o

main.o : main.c fmb3dt.h ../Frame/frame.h Makefile
	$(COMPILER) -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb3dt.o frame.o Makefile
	$(COMPILER) -o unitTests unitTests.o fmb3dt.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb3dt.h ../Frame/frame.h Makefile
	$(COMPILER) -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb3dt.o sat.o frame.o Makefile
	$(COMPILER) -o validation validation.o fmb3dt.o sat.o frame.o

validation.o : validation.c fmb3dt.h ../SAT/sat.h ../Frame/frame.h Makefile
	$(COMPILER) -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb3dt.o sat.o frame.o Makefile
	$(COMPILER) -o qualification qualification.o fmb3dt.o sat.o frame.o
	    $(LINK_ARG)

qualification.o : qualification.c fmb3dt.h ../SAT/sat.h ../Frame/frame.h
    Makefile
	$(COMPILER) -c qualification.c $(BUILD_ARG)

fmb3dt.o : fmb3dt.c fmb3dt.h ../Frame/frame.h Makefile
	$(COMPILER) -c fmb3dt.c $(BUILD_ARG)

sat.o : ../SAT/sat.c ../SAT/sat.h ../Frame/frame.h Makefile
	$(COMPILER) -c ../SAT/sat.c $(BUILD_ARG)

frame.o : ../Frame/frame.c ../Frame/frame.h Makefile
	$(COMPILER) -c ../Frame/frame.c $(BUILD_ARG)

clean :
	rm -f *.o main unitTests validation qualification

valgrind :
	valgrind -v --track-origins=yes --leak-check=full \
	--gen-suppressions=yes --show-leak-kinds=all ./main
```

References

[1] J.J.-B. Fourier. Oeuvres II. Paris, 1890

[2] T.S. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen.* Thesis, 1936. Reprinted in: *Theodore S. Motzkin: selected papers* (D.Cantor et al., eds,), Birkhäuser, Boston, 1983.