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

P. Baillehache

December 30, 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:

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

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

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:

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

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

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

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

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:

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

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

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

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:

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

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

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

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

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

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

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

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 the cases 2D and 3D.

4.1 2D static

`algo`

4.2 3D static

`algo`

4.3 2D dynamic

`algo`

4.4 3D dynamic

`algo`

5 Implementation

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

5.1 Frames

5.1.1 Header

```
#ifndef __FRAME_H_
#define __FRAME_H_

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

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

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

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

typedef enum {
  FrameCuboid,
  FrameTetrahedron
} FrameType;

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

// Axis aligned bounding box structure
typedef struct {
  // 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] = -1.0 * tc[0][1] / det;
  tic[1][0] = -1.0 * tc[1][0] / det;
  tic[1][1] = tc[0][0] / det;

}

void Frame3DUpdateInv(Frame3D* const that) {

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

  // Update the inverse components
  double det =
    tc[0][0] * (tc[1][1] * tc[2][2]- tc[1][2] * tc[2][1]) -
    tc[1][0] * (tc[0][1] * tc[2][2]- tc[0][2] * tc[2][1]) +
    tc[2][0] * (tc[0][1] * tc[1][2]- tc[0][2] * tc[1][1]);
```

```
  if (fabs(det) < EPSILON) {
    fprintf(stderr,
      "FrameUpdateInv: det == 0.0\n");
    exit(1);
  }

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

}

// 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] = -1.0 * tc[0][1] / det;
  tic[1][0] = -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 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 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
//printf("inconsistent %.9f %.9f\n",Yp[*nbRemainRows], sumNegCoeff + EPSILON
    );
          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 < -1.0 * EPSILON) {

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

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

        // Update the minimum bound
        *min = y;

      }

    }

  }

}

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

```
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection2D(
  const Frame2D* const that,
  const Frame2D* const tho,
         AABB2D* const bdgBox) {
//Frame2DPrint(that);printf("\n");
//Frame2DPrint(tho);printf("\n");
  // Get the projection of the Frame 'tho' in Frame 'that' coordinates
  // system
  Frame2D thoProj;
  Frame2DImportFrame(that, tho, &thoProj);

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

  // Create the inequality system

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

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

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

  if (that->type == FrameCuboid) {

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

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

  } else {

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

```
}

if (tho->type == FrameCuboid) {

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

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

} else {

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

}

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

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

// Solve the system

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

// Declare variables to eliminate the first variable
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;

  }

  // 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;

  }

  // 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;
```

```
  }

  // If we've reached here the two Frames are intersecting
//printf("inter\n");
  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 < -1.0 * EPSILON) {

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

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

        // Update the minimum bound
        *min = y;

      }

    }

  }

}

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

```
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection3D(
  const Frame3D* const that,
  const Frame3D* const tho,
         AABB3D* const bdgBox) {

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

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

  // Create the inequality system

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

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

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

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

  if (that->type == FrameCuboid) {

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

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

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

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

} else {

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

}

if (tho->type == FrameCuboid) {

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

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

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

} else {

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

}
```

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

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

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

// Solve the system

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

// Declare variables to eliminate the first variable
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 < -1.0 * EPSILON) {

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

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

        // Update the minimum bound
        *min = y;

      }

    }

  }

}

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
// The resulting AABB is given in 'tho' 's local coordinates system
bool FMBTestIntersection2DTime(
  const Frame2DTime* const that,
  const Frame2DTime* const tho,
         AABB2DTime* const bdgBox) {

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

  // Declare two variables to memorize the system to be solved M.X <= Y
  // (M arrangement is [iRow][iCol])
  double M[10][3];
  double Y[10];

  // Create the inequality system

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

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

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

if (that->type == FrameCuboid) {

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

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

} else {

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

}

if (tho->type == FrameCuboid) {

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

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

} else {
```

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

}

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

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

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

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

// Solve the system

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

// Declare variables to eliminate the first variable
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 < -1.0 * EPSILON) {

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

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

        // Update the minimum bound
        *min = y;

      }

    }

  }

}

// Test for intersection between Frame 'that' and Frame 'tho'
// Return true if the two Frames are intersecting, else false
// If the Frame are intersecting the AABB of the intersection
// is stored into 'bdgBox', else 'bdgBox' is not modified
// If 'bdgBox' is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
// 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.177)

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
```

```c
#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.177)

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]-
```

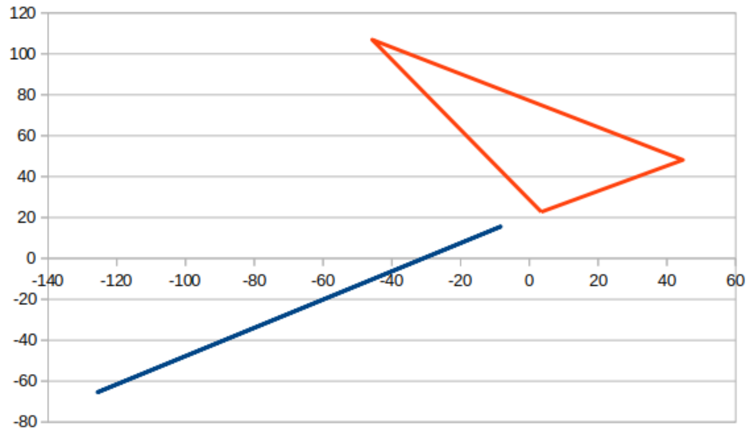
```
      paramP.comp[0][2] * paramP.comp[1][1]);

    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 470452 intersections and 1529454 no intersections
```

8.2.3 2D dynamic

```
===== 2D dynamic ======
Validation2DTime has succeed.
Tested 744114 intersections and 1255804 no intersections
```

8.2.4 3D static

```
===== 3D static ======
Validation3D has succeed.
Tested 315158 intersections and 1684842 no intersections
```

8.2.5 3D dynamic

```
===== 3D dynamic ======
Validation3DTime has succeed.
Tested 523450 intersections and 1476550 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
```

```c
#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>
#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;
}
```

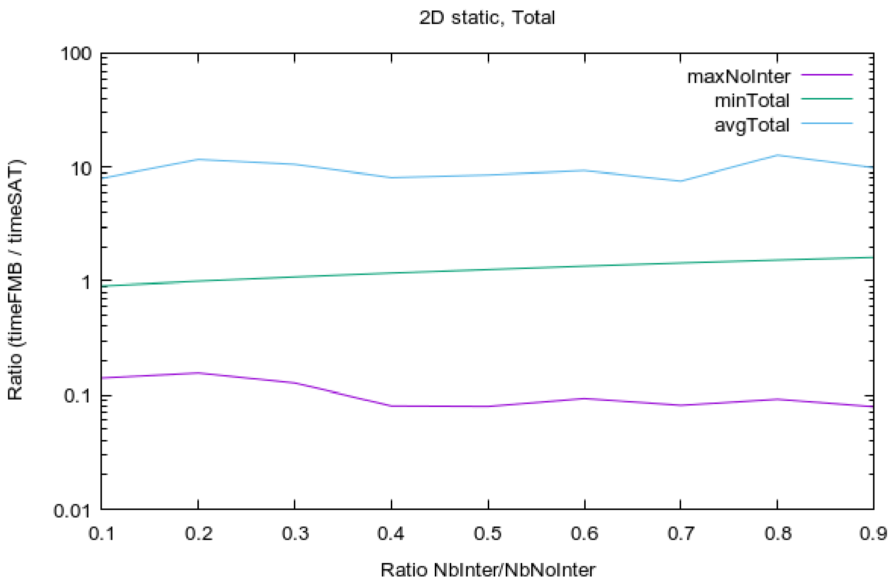
9.2 Results

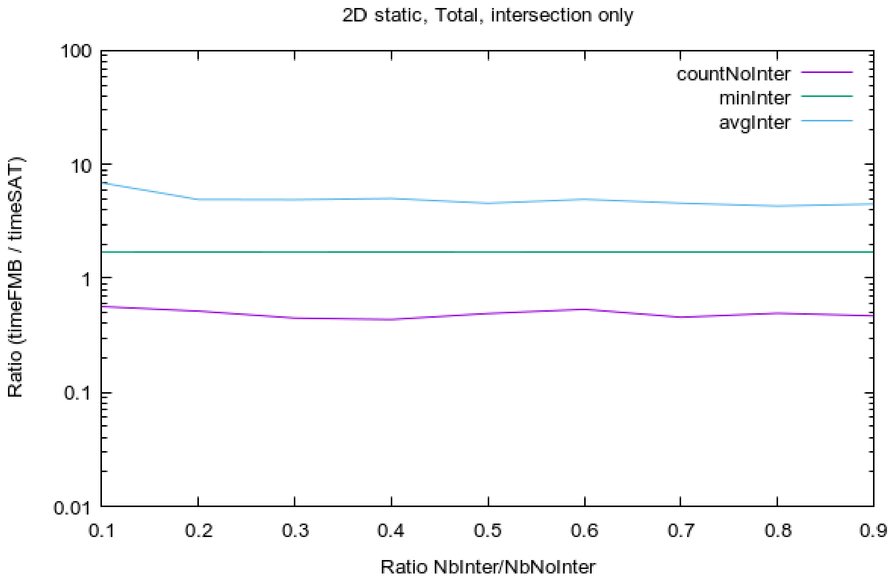
9.2.1 2D static

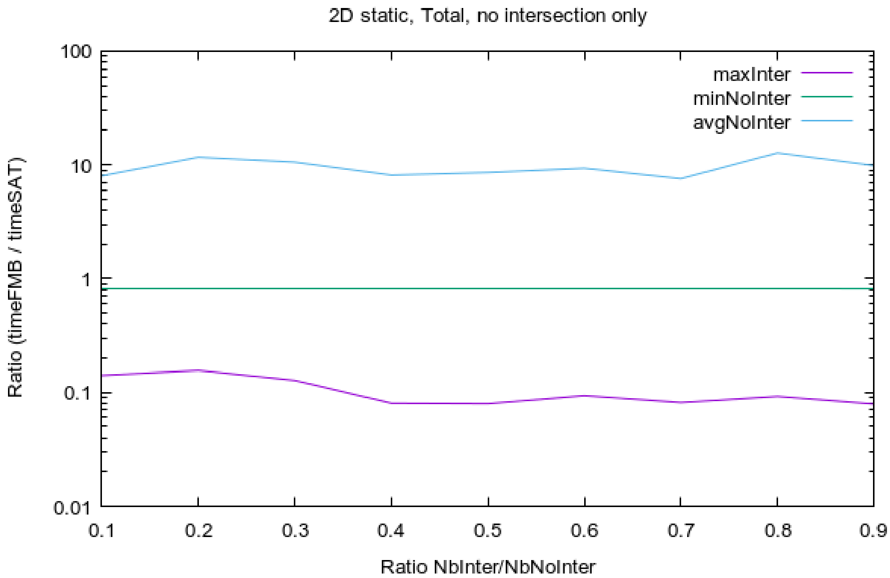
```
ratio Inter/NoInter      countInter      countNoInter    minInter
    avgInter        maxInter        minNoInter      avgNoInter
    maxNoInter      minTotal        avgTotal        maxTotal
    countInterCC    countNoInterCC  minInterCC      avgInterCC
    maxInterCC      minNoInterCC    avgNoInterCC    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     46966   153022  0.566434        1.703869        6.923077
    0.141026        0.819928        8.000000        0.141026        0.908322
            8.000000        13244   36370   0.880597        2.180478
    6.923077        0.156627        0.742749        5.761905        0.156627
            0.886522        6.923077        11680   38422   0.756098
    1.662513        3.576271        0.157895        0.826459        8.000000
            0.157895        0.910064        8.000000        11638   38122
    0.728682        1.663660        4.275862        0.144444        0.824793
            7.000000        0.144444        0.908679        7.000000
    10404   40108   0.566434        1.188565        2.393939        0.141026
            0.879033        7.687500        0.141026        0.909986
    7.687500
0.2     46868   153118  0.515924        1.704268        4.927273
    0.155844        0.819394        11.666667       0.155844        0.996369
            11.666667       12990   36922   0.881481        2.181347
    4.927273        0.180556        0.742780        5.575000        0.180556
            1.030494        5.575000        11736   38144   0.679487
    1.664811        3.465517        0.155844        0.828881        8.375000
            0.155844        0.996067        8.375000        11926   38114
    0.635762        1.663998        3.770492        0.175676        0.825917
            11.666667       0.175676        0.993533        11.666667
    10216   39938   0.515924        1.189984        2.402985        0.160494
            0.874937        6.200000        0.160494        0.937946
    6.200000
0.3     46534   153454  0.447514        1.702537        4.905660
    0.127451        0.819496        10.562500       0.127451        1.084409
            10.562500       12904   37164   0.906433        2.181373
    4.905660        0.170732        0.747585        4.590909        0.170732
            1.177722        4.905660        11822   38556   0.587629
```

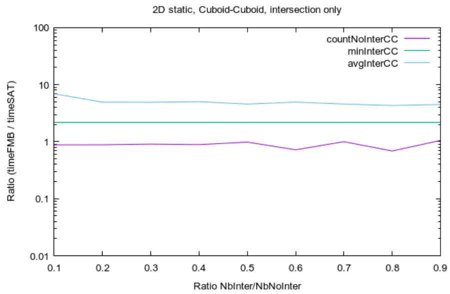
```
    1.662497        3.216667        0.127451        0.822627        6.533333
            0.127451        1.074588        6.533333        11592   38250
    0.511737        1.662599        3.453125        0.151685        0.830219
            10.562500       0.151685        1.079933        10.562500
    10216   39484   0.447514        1.189363        2.454545        0.141026
            0.873737        8.066667        0.141026        0.968425
    8.066667
0.4     47032   152964  0.434783        1.705222        5.037037
    0.080460        0.819058        8.125000        0.080460        1.173524
            8.125000        13234   36748   0.887640        2.179615
    5.037037        0.160494        0.745819        4.465116        0.160494
            1.319337        5.037037        11906   38080   0.519231
    1.663047        3.716667        0.080460        0.824219        8.125000
            0.080460        1.159750        8.125000        11622   38348
    0.697368        1.663416        4.114754        0.192308        0.823624
            6.533333        0.192308        1.159541        6.533333
    10270   39788   0.434783        1.190119        2.614286        0.101695
            0.877361        7.866667        0.101695        1.002464
    7.866667
0.5     47012   152976  0.490909        1.702165        4.563636
    0.079710        0.819014        8.562500        0.079710        1.260590
            8.562500        13072   36742   0.983471        2.179741
    4.563636        0.155844        0.747984        6.023256        0.155844
            1.463862        6.023256        11778   38062   0.644295
    1.663080        3.672131        0.125000        0.824957        6.533333
            0.125000        1.244018        6.533333        11736   38620
    0.610063        1.665692        3.612903        0.079710        0.825705
            8.562500        0.079710        1.245698        8.562500
    10426   39552   0.490909        1.188596        2.895522        0.162500
            0.872745        6.266667        0.162500        1.030671
    6.266667
0.6     47002   152988  0.534759        1.704474        4.945455
    0.093525        0.819909        9.333333        0.093525        1.350648
            9.333333        13292   36688   0.721053        2.180593
    4.945455        0.093525        0.748104        4.950000        0.093525
            1.607597        4.950000        11654   38232   0.583333
    1.663303        4.644068        0.142857        0.827837        6.466667
            0.142857        1.329116        6.466667        11648   38364
    0.700730        1.663686        4.271186        0.147727        0.826859
            8.687500        0.147727        1.328955        8.687500
    10408   39704   0.534759        1.188170        2.880597        0.171875
            0.871911        9.333333        0.171875        1.061667
    9.333333
0.7     47202   152792  0.455056        1.706769        4.576923
    0.081481        0.819624        7.562500        0.081481        1.440625
            7.562500        13362   36626   1.000000        2.181935
    4.576923        0.081481        0.743440        7.562500        0.081481
            1.750386        7.562500        11694   38058   0.477477
    1.662745        3.767123        0.176471        0.823407        7.133333
            0.176471        1.410944        7.133333        11892   38440
    0.715328        1.662450        3.573770        0.129412        0.829590
            6.625000        0.129412        1.412592        6.625000
    10254   39668   0.455056        1.189185        2.985075        0.139785
            0.876679        7.117647        0.139785        1.095433
    7.117647
0.8     46816   153182  0.493902        1.707018        4.314815
    0.092199        0.819990        12.733333       0.092199        1.529612
            12.733333       13322   36370   0.687861        2.180205
    4.314815        0.092199        0.742926        3.833333        0.092199
            1.892749        4.314815        11494   38566   0.646259
    1.662133        3.333333        0.117117        0.827329        6.466667
            0.117117        1.495172        6.466667        11836   38062
```

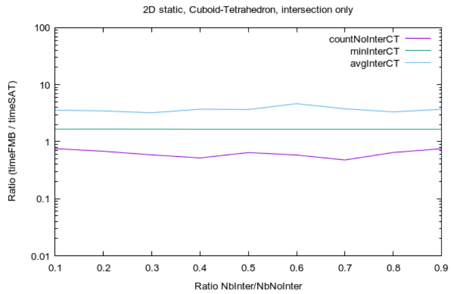
0.572289 1.663755 4.081967 0.112000 0.825611
6.533333 0.112000 1.496127 6.533333
10164 40184 0.493902 1.187947 2.911765 0.159420
0.877372 12.733333 0.159420 1.125832
12.733333
0.9 46924 153070 0.468571 1.704683 4.500000
0.079137 0.817229 9.875000 0.079137 1.615938
9.875000 13396 37030 1.051724 2.179104
4.500000 0.090909 0.742654 4.952381 0.090909
2.035459 4.952381 11522 38170 0.753968
1.661215 3.694915 0.079137 0.823619 6.466667
0.079137 1.577456 6.466667 11654 38286
0.903448 1.662193 3.629032 0.173913 0.823679
9.875000 0.173913 1.578341 9.875000
10352 39584 0.468571 1.186974 3.000000 0.164557
0.874594 8.500000 0.164557 1.155736
8.500000

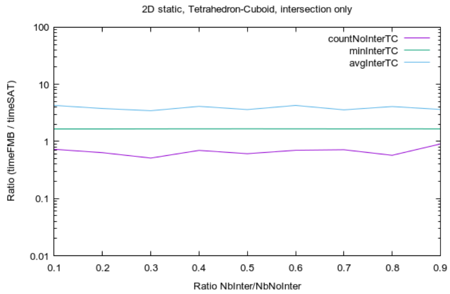


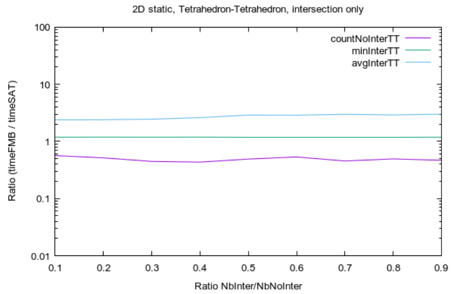

2D static, Total, intersection only
countNoInter
minInter
avgInter
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter

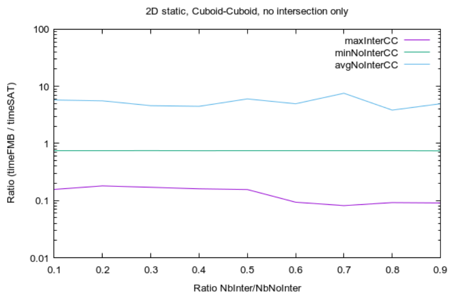

2D static, Total, no intersection only
maxInter
minNoInter
avgNoInter
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter

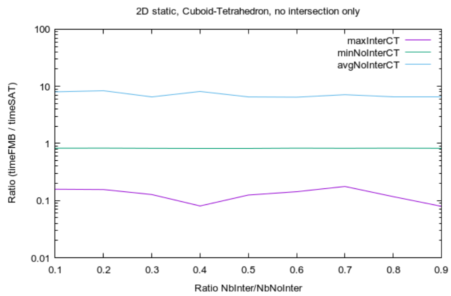


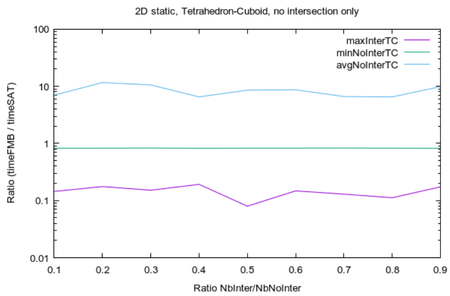


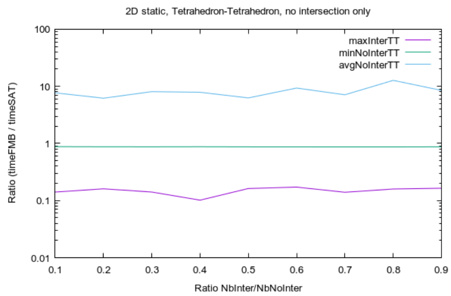








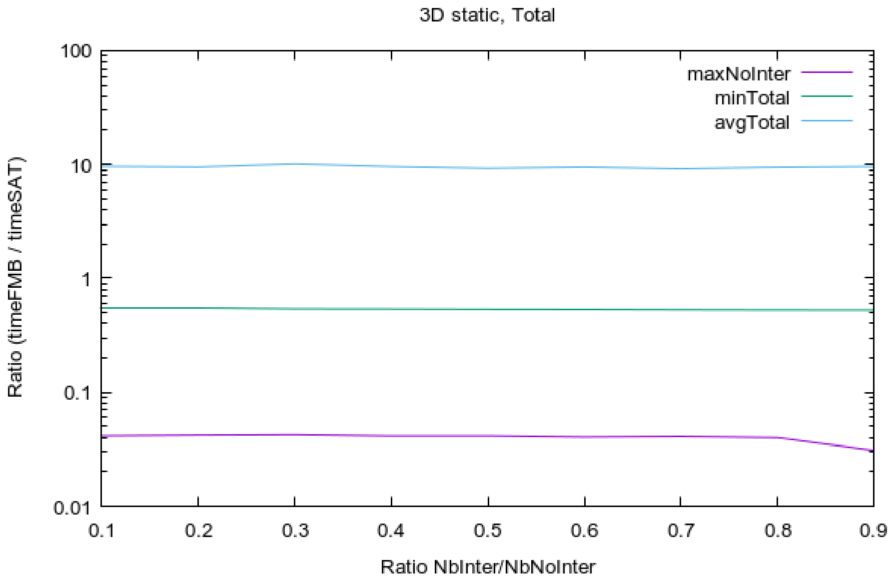


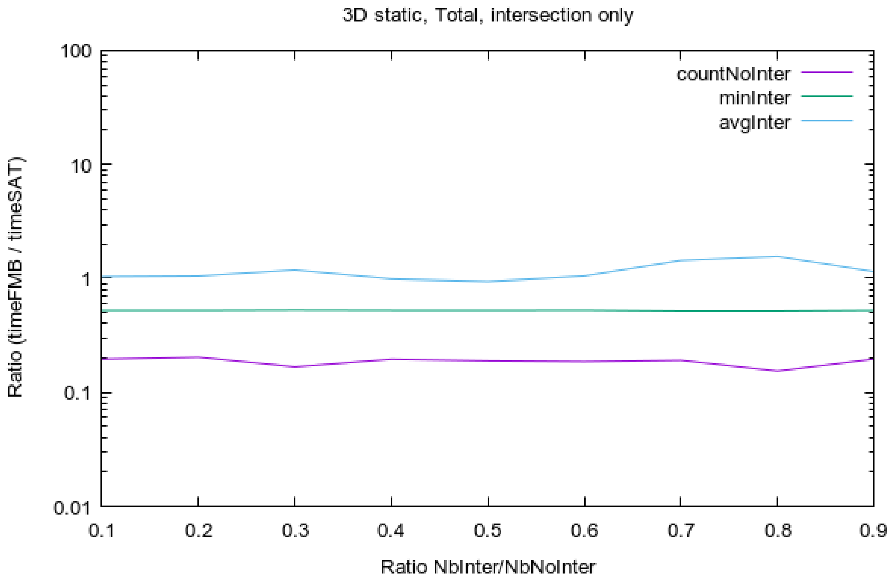


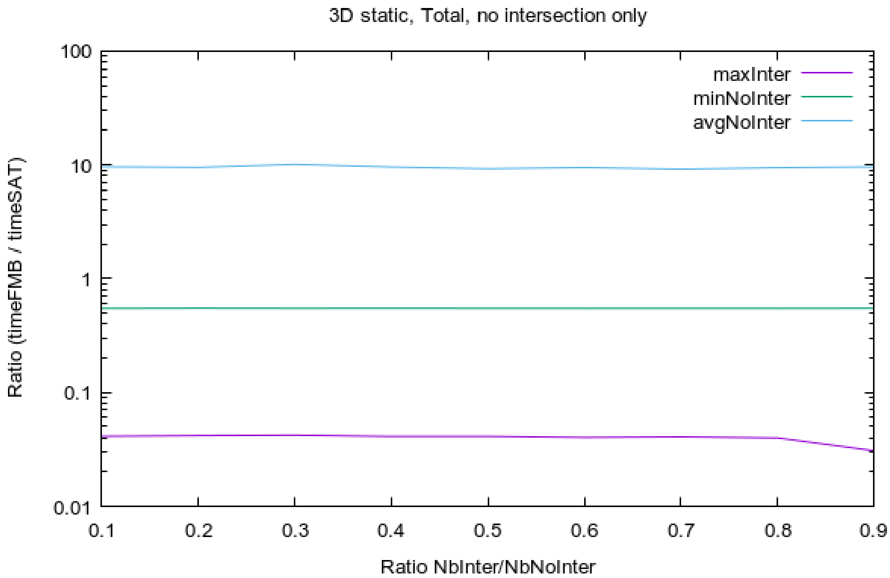
9.2.2 3D static

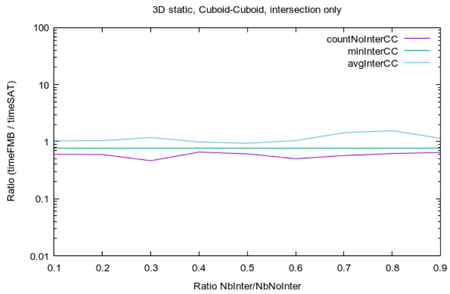
```
ratio Inter/NoInter     countInter      countNoInter    minInter
    avgInter        maxInter        minNoInter      avgNoInter
    maxNoInter      minTotal        avgTotal        maxTotal
    countInterCC    countNoInterCC  minInterCC      avgInterCC
    maxInterCC      minNoInterCC    avgNoInterCC    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     31716   168284  0.195513        0.525441        1.031293
    0.041359        0.545129        9.612903        0.041359        0.543160
            9.612903        10616   39390   0.602136        0.778821
    1.031293        0.063559        0.387620        3.034014        0.063559
            0.426740        3.034014        7920    42174   0.218111
    0.445642        0.780516        0.046129        0.529435        9.612903
            0.046129        0.521056        9.612903        8042    41840
    0.291444        0.444923        0.645333        0.046774        0.530002
            9.468750        0.046774        0.521494        9.468750
    5138    44880   0.195513        0.250948        0.304242        0.041359
            0.712220        8.560000        0.041359        0.666093
    8.560000
0.2     31770   168230  0.203681        0.524833        1.045620
    0.041916        0.547554        9.500000        0.041916        0.543010
            9.500000        10710   39298   0.595007        0.778798
    1.045620        0.064444        0.391099        3.013699        0.064444
            0.468639        3.013699        7878    41378   0.279503
    0.445876        0.708029        0.045381        0.529847        9.500000
            0.045381        0.513053        9.500000        7796    42444
    0.285714        0.445254        0.709353        0.047776        0.531599
            9.437500        0.047776        0.514330        9.437500
    5386    45110   0.203681        0.250499        0.378517        0.041916
            0.715104        8.666667        0.041916        0.622183
    8.666667
0.3     31592   168408  0.167113        0.527451        1.181501
    0.042296        0.545777        10.093750       0.042296        0.540279
            10.093750       10744   39560   0.463895        0.779107
    1.181501        0.065022        0.390479        3.166667        0.065022
            0.507067        3.166667        8020    41950   0.276243
    0.445714        0.709251        0.042296        0.534041        9.757576
            0.042296        0.507543        9.757576        7730    42384
    0.244944        0.444968        0.691971        0.047386        0.526136
            10.093750       0.047386        0.501785        10.093750
    5098    44514   0.167113        0.250738        0.383526        0.043011
            0.713552        8.833333        0.043011        0.574708
    8.833333
0.4     31716   168284  0.195376        0.525420        0.987273
    0.041237        0.546552        9.580645        0.041237        0.538099
            9.580645        10728   39340   0.656832        0.778962
    0.987273        0.064444        0.390907        2.993151        0.064444
            0.546129        2.993151        7832    42372   0.270627
    0.445376        0.799423        0.045902        0.528674        9.580645
            0.045902        0.495355        9.580645        7822    41934
    0.266423        0.445009        0.720588        0.047002        0.531795
            9.437500        0.047002        0.497081        9.437500
    5334    44638   0.195376        0.250931        0.424710        0.041237
            0.714558        8.958333        0.041237        0.529108
    8.958333
```

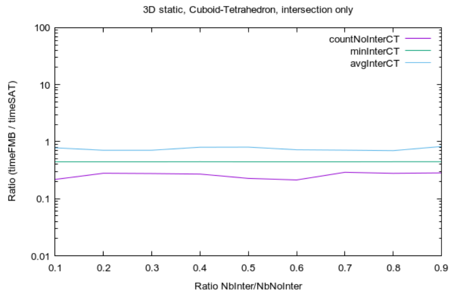
```
0.5     31526   168474  0.188960        0.524791        0.937276
    0.041298        0.545450        9.242424        0.041298        0.535120
        9.242424        10556   39614   0.611272        0.778890
    0.937276        0.065022        0.391878        3.337838        0.065022
        0.585384        3.337838        7842    42412   0.227149
    0.445859        0.803184        0.043956        0.527911        9.242424
        0.043956        0.486885        9.242424        7898    42110
    0.266509        0.445011        0.729844        0.046802        0.531517
        9.000000        0.046802        0.488264        9.000000
    5230    44338   0.188960        0.250761        0.345806        0.041298
        0.712667        8.833333        0.041298        0.481714
    8.833333
0.6     31826   168174  0.185787        0.526009        1.046595
    0.040346        0.545207        9.468750        0.040346        0.533688
        9.468750        10752   39380   0.502924        0.778944
    1.046595        0.064444        0.391719        3.119205        0.064444
        0.624054        3.119205        7708    41896   0.213256
    0.445803        0.721481        0.045089        0.531062        9.468750
        0.045089        0.479906        9.468750        8104    42324
    0.283908        0.445171        0.738416        0.048093        0.525634
        9.181818        0.048093        0.477356        9.181818
    5262    44574   0.185787        0.251172        0.466431        0.040346
        0.712689        9.000000        0.040346        0.435779
    9.000000
0.7     32002   167998  0.191130        0.523725        1.436364
    0.040816        0.545714        9.156250        0.040816        0.530322
        9.156250        10656   39184   0.571429        0.779126
    1.436364        0.065611        0.390225        3.413043        0.065611
        0.662456        3.413043        7926    41932   0.290451
    0.445650        0.711340        0.044554        0.531201        9.156250
        0.044554        0.471315        9.156250        8018    42184
    0.246440        0.445298        0.769118        0.046624        0.529724
        8.911765        0.046624        0.470626        8.911765
    5402    44698   0.191130        0.250883        0.326223        0.040816
        0.710726        8.640000        0.040816        0.388836
    8.640000
0.8     31878   168122  0.153328        0.523698        1.559415
    0.039943        0.544468        9.437500        0.039943        0.527852
        9.437500        10608   39340   0.619392        0.778926
    1.559415        0.062907        0.391076        3.142857        0.062907
        0.701356        3.142857        7870    42412   0.278416
    0.445469        0.696925        0.043818        0.529756        9.250000
        0.043818        0.462327        9.250000        8040    42042
    0.229990        0.445315        1.159590        0.046400        0.526192
        9.437500        0.046400        0.461490        9.437500
    5360    44328   0.153328        0.251012        0.347222        0.039943
        0.712010        8.916667        0.039943        0.343212
    8.916667
0.9     31928   168072  0.195789        0.524508        1.144177
    0.030818        0.546272        9.593750        0.030818        0.526684
        9.593750        10650   39294   0.646526        0.778930
    1.144177        0.065611        0.392040        3.187919        0.065611
        0.740241        3.187919        8026    42252   0.283505
    0.445702        0.823358        0.044800        0.529001        9.593750
        0.044800        0.454032        9.593750        7984    41854
    0.282636        0.444964        0.650442        0.030818        0.528834
        9.454545        0.030818        0.453351        9.454545
    5268    44672   0.195789        0.250776        0.387255        0.040580
        0.714611        8.916667        0.040580        0.297160
    8.916667
```

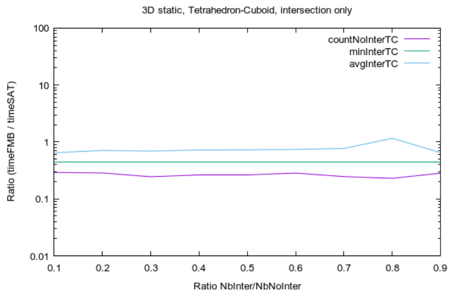

3D static, Total
100
10
1
0.1
0.01
Ratio (timeFMB / timeSAT)
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio NbInter/NbNoInter
maxNoInter
minTotal
avgTotal

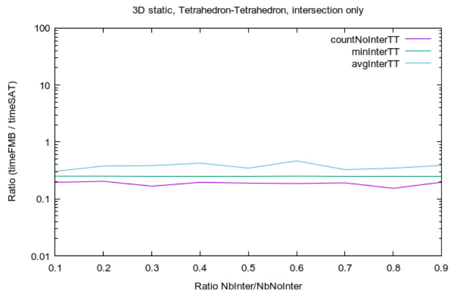

3D static, Total, intersection only
countNoInter
minInter
avgInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio NbInter/NbNoInter

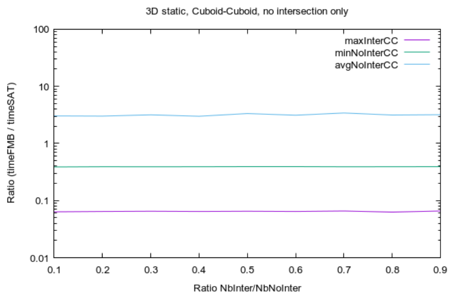

3D static, Total, no intersection only
maxInter
minNoInter
avgNoInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio NbInter/NbNoInter

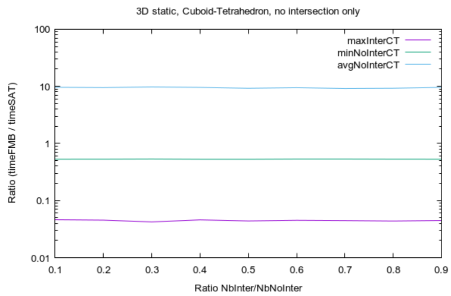


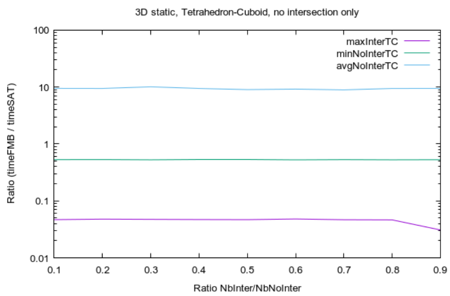


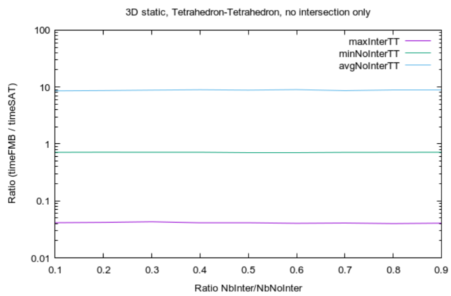








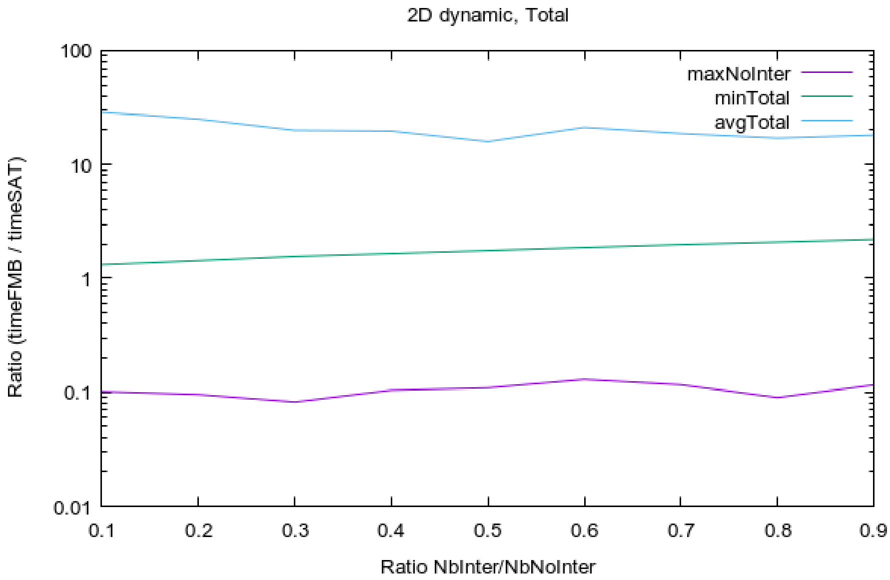


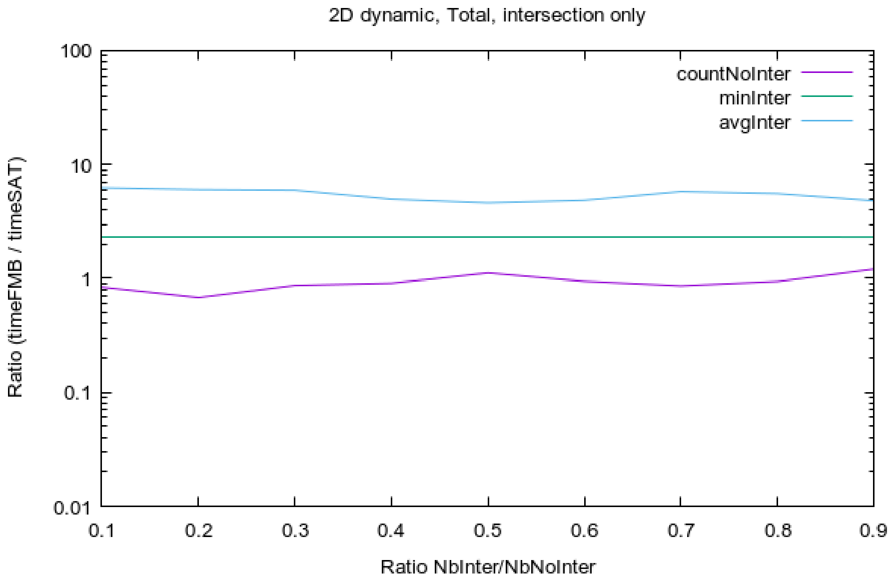


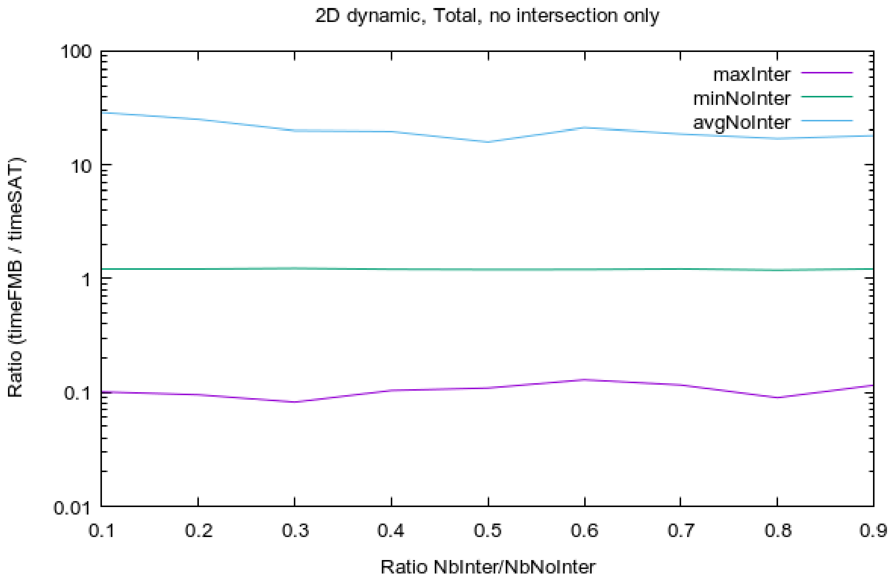
9.2.3 2D dynamic

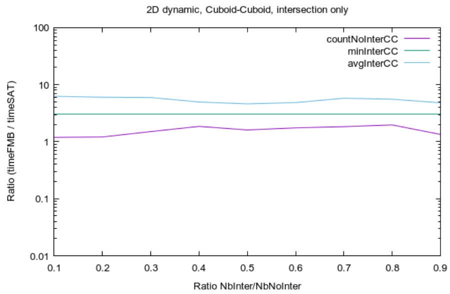
```
ratio Inter/NoInter     countInter      countNoInter    minInter
    avgInter        maxInter        minNoInter      avgNoInter
    maxNoInter      minTotal        avgTotal        maxTotal
    countInterCC    countNoInterCC  minInterCC      avgInterCC
    maxInterCC      minNoInterCC    avgNoInterCC    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     74120   125872  0.837061        2.311511        6.248062
    0.101266        1.208643        28.800000       0.101266        1.318930
            28.800000       19700   30536   1.191892        3.028708
    6.248062        0.103896        1.218248        25.115385       0.103896
            1.399294        25.115385       18492   31106   0.974763
    2.251752        5.408163        0.121429        1.221026
    20.652174       0.121429        1.324099        20.652174       18898
    31704   1.046099        2.248574        4.803150        0.101266
    1.205695        28.800000       0.101266        1.309983
    28.800000       17030   32526   0.837061        1.616598        3.189349
            0.106918        1.190658        15.791667       0.106918
    1.233252        15.791667
0.2     74344   125642  0.680412        2.313088        6.036496
    0.095506        1.209531        25.071429       0.095506        1.430242
            25.071429       19826   29896   1.210811        3.032327
    6.036496        0.106918        1.203857        22.629630       0.106918
            1.569551        22.629630       18696   31758   0.938907
    2.253203        4.496063        0.099415        1.214103
    25.071429       0.099415        1.421923        25.071429       18690
    31354   1.035587        2.248050        4.658730        0.112676
    1.237143        18.958333       0.112676        1.439324
    18.958333       17132   32634   0.680412        1.617054        2.936306
            0.095506        1.183751        14.166667       0.095506
    1.270412        14.166667
0.3     74496   125496  0.867987        2.313805        5.951049
    0.082126        1.234912        20.000000       0.082126        1.558579
            20.000000       19888   30258   1.504065        3.031555
    5.951049        0.093923        1.225522        20.000000       0.093923
            1.767332        20.000000       18934   31208   0.948220
    2.251892        4.622047        0.082126        1.254529
    19.434783       0.082126        1.553738        19.434783       18608
    31392   0.996865        2.248619        3.976923        0.098837
    1.259352        18.272727       0.098837        1.556132
    18.272727       17066   32638   0.867987        1.617133        2.811321
            0.108280        1.201351        17.037037       0.108280
    1.326086        17.037037
0.4     74248   125740  0.907216        2.310234        4.967213
    0.104167        1.211019        19.720000       0.104167        1.650705
            19.720000       19874   29828   1.855721        3.027625
    4.967213        0.125926        1.214654        16.653846       0.125926
            1.939843        16.653846       18614   31490   1.276119
    2.249098        4.297297        0.104167        1.224775
    19.720000       0.104167        1.634504        19.720000       18630
    31458   1.200000        2.246232        4.617886        0.117241
    1.222226        17.333333       0.117241        1.631829
    17.333333       17130   32964   0.907216        1.613965        3.316770
            0.129496        1.183895        14.040000       0.129496
    1.355923        14.040000
```

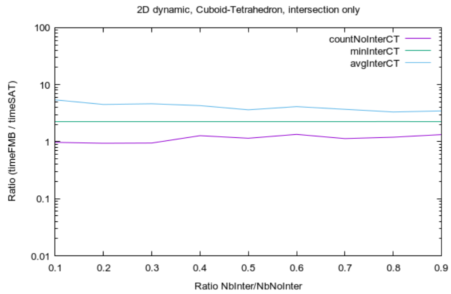
```
0.5     74084   125908  1.115880        2.306832        4.608333
    0.109589        1.202898        15.923077       0.109589        1.754865
        15.923077       19638   30158   1.605578        3.025066
    4.608333        0.126050        1.204034        15.923077       0.126050
        2.114550        15.923077       18646   31918   1.149826
    2.248365        3.625954        0.141509        1.228429
    15.727273       0.141509        1.738397        15.727273       18658
    31424   1.286996        2.245789        3.446154        0.119718
    1.202380        15.304348       0.119718        1.724085
    15.304348       17142   32408   1.115880        1.614057        2.380645
        0.109589        1.177196        13.387097       0.109589
    1.395627        13.387097
0.6     74746   125246  0.942238        2.307949        4.851240
    0.129630        1.205642        21.200000       0.129630        1.867027
        21.200000       19974   30126   1.748815        3.026642
    4.851240        0.137615        1.196891        19.555556       0.137615
        2.294741        19.555556       18478   31238   1.344037
    2.247793        4.123077        0.147059        1.229855
    21.200000       0.147059        1.840618        21.200000       18880
    31366   1.336406        2.246459        4.765625        0.149533
    1.206451        18.538462       0.149533        1.830456
    18.538462       17414   32516   0.942238        1.614102        2.830508
        0.129630        1.189710        17.307692       0.129630
    1.444345        17.307692
0.7     74358   125630  0.859935        2.305549        5.787402
    0.116438        1.219713        18.739130       0.116438        1.979798
        18.739130       19884   29816   1.836634        3.025961
    5.787402        0.164835        1.240444        16.840000       0.164835
        2.490306        16.840000       18782   31532   1.134387
    2.246623        3.696970        0.119048        1.233620
    18.739130       0.119048        1.942722        18.739130       18140
    31262   1.469027        2.245901        4.664179        0.116438
    1.223522        15.727273       0.116438        1.939187
    15.727273       17552   33020   0.859935        1.614124        3.786070
        0.133333        1.184107        11.653846       0.133333
    1.485119        11.653846
0.8     73968   126030  0.938628        2.306300        5.556452
    0.089947        1.190794        17.086957       0.089947        2.083199
        17.086957       19676   30430   1.967914        3.025384
    5.556452        0.108974        1.169921        15.592593       0.108974
        2.654291        15.592593       18820   31686   1.200000
    2.247608        3.320000        0.089947        1.205285
    17.086957       0.089947        2.039143        17.086957       18230
    31550   1.414634        2.245347        3.679688        0.148148
    1.188884        15.681818       0.148148        2.034055
    15.681818       17242   32364   0.938628        1.614215        2.168750
        0.131579        1.198094        13.640000       0.131579
    1.530991        13.640000
0.9     74498   125494  1.204762        2.311647        4.806723
    0.115942        1.219154        18.153846       0.115942        2.202398
        18.153846       20168   30096   1.340580        3.026141
    4.806723        0.205128        1.234743        18.153846       0.205128
        2.847002        18.153846       18534   31602   1.331897
    2.247253        3.460938        0.119718        1.237502
    15.545455       0.119718        2.146278        15.545455       18626
    31270   1.459144        2.245886        3.460938        0.119048
    1.230062        15.458333       0.119048        2.144303
    15.458333       17170   32526   1.204762        1.613246        2.692308
        0.115942        1.176418        15.208333       0.115942
    1.569563        15.208333
```

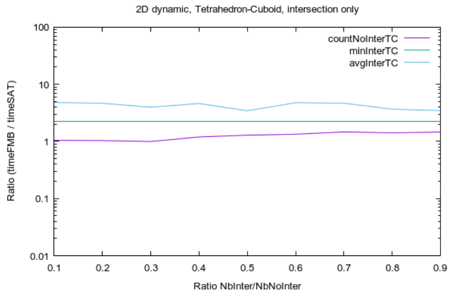

2D dynamic, Total
maxNoInter
minTotal
avgTotal
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio NbInter/NbNoInter

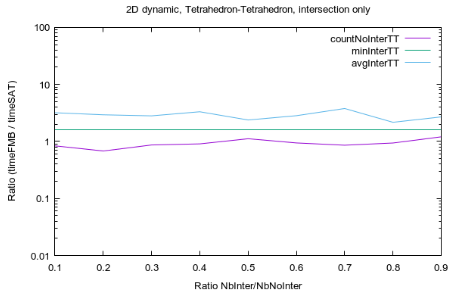

2D dynamic, Total, intersection only
countNoInter
minInter
avgInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio NbInter/NbNoInter

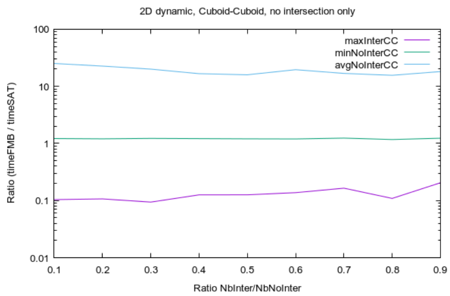

2D dynamic, Total, no intersection only
maxInter
minNoInter
avgNoInter
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio NbInter/NbNoInter

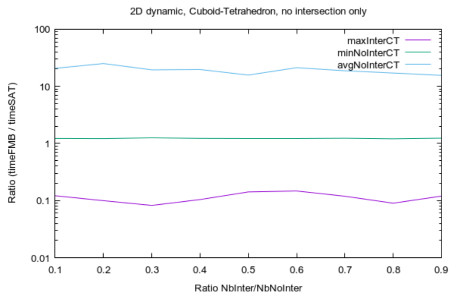


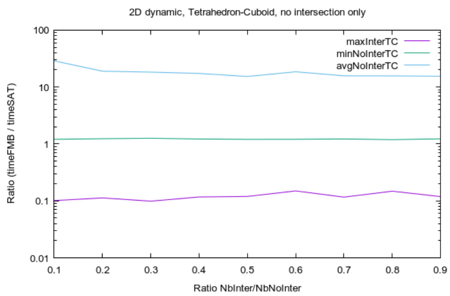


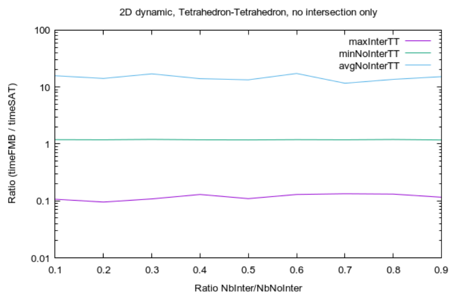








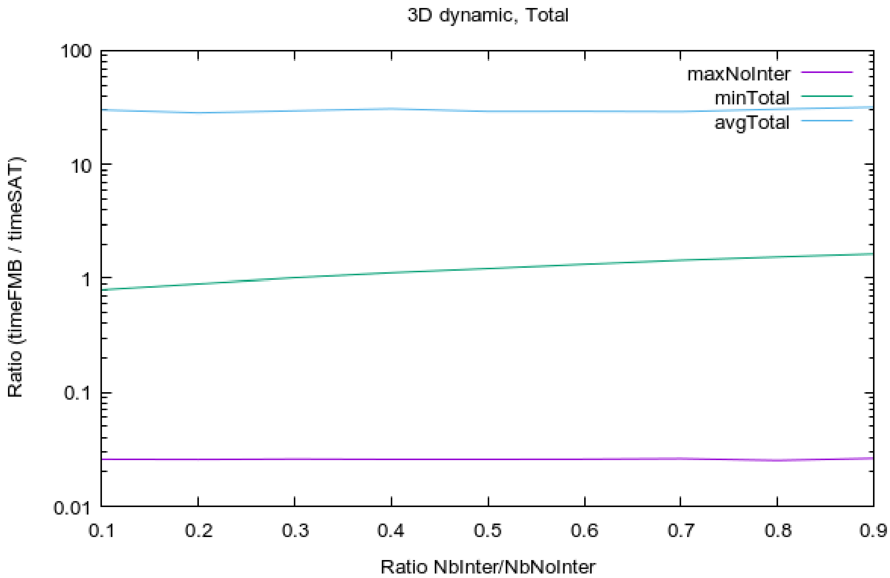


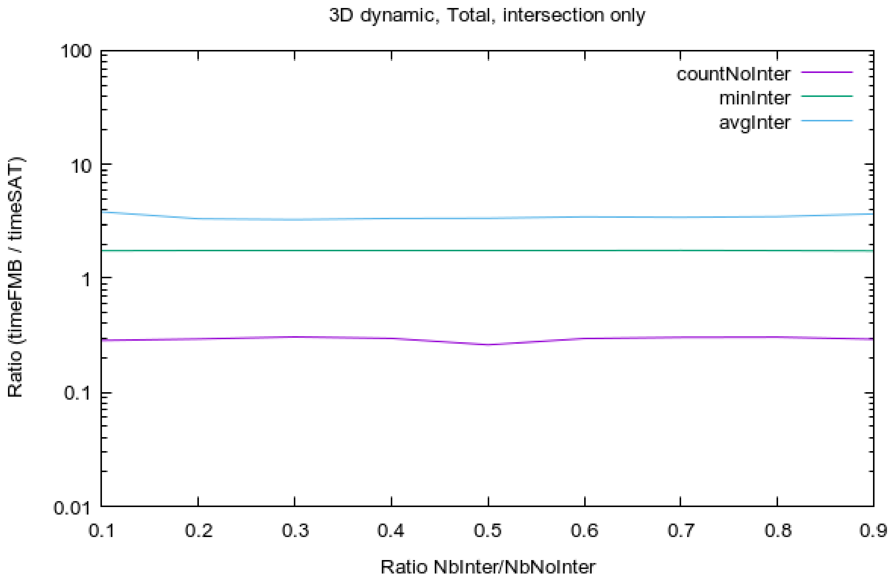


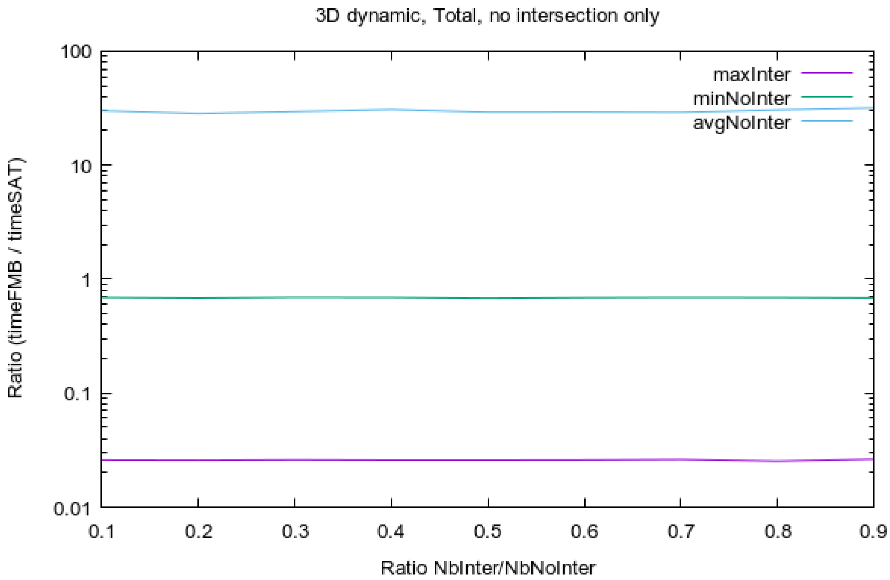
9.2.4 3D dynamic

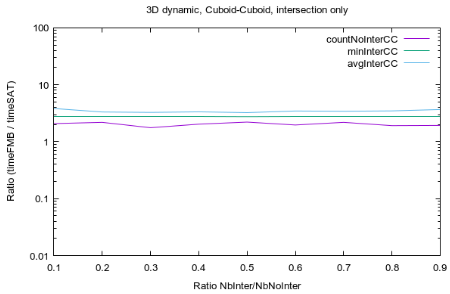
```
ratio Inter/NoInter     countInter      countNoInter    minInter
    avgInter        maxInter        minNoInter      avgNoInter
    maxNoInter      minTotal        avgTotal        maxTotal
    countInterCC    countNoInterCC  minInterCC      avgInterCC
    maxInterCC      minNoInterCC    avgNoInterCC    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     52368   147632  0.286049        1.749111        3.827556
    0.025856        0.690933        30.054054       0.025856        0.796751
            30.054054       15878   33932   2.079092        2.759798
    3.827556        0.038136        0.591372        13.620833       0.038136
            0.808214        13.620833       13050   36746   0.434733
    1.587665        3.167442        0.025856        0.685968
    24.448980       0.025856        0.776138        24.448980       13252
    37254   0.471136        1.590276        3.125675        0.026583
    0.702744        25.396226       0.026583        0.791497
    25.396226       10188   39700   0.286049        0.587359        0.990444
            0.026154        0.769541        30.054054       0.026154
    0.751323        30.054054
0.2     52346   147654  0.295306        1.758523        3.329092
    0.025738        0.681488        28.333333       0.025738        0.896895
            28.333333       16152   33910   2.187500        2.759516
    3.329092        0.037815        0.581714        13.888889       0.037815
            1.017274        13.888889       13064   37180   0.464417
    1.590671        3.155093        0.027844        0.686945
    21.886792       0.027844        0.867691        21.886792       13048
    36526   0.480315        1.592447        3.104006        0.027174
    0.688474        20.169492       0.027174        0.869269
    20.169492       10082   40038   0.295306        0.587299        0.971429
            0.025738        0.754548        28.333333       0.025738
    0.721098        28.333333
0.3     52436   147562  0.307540        1.754851        3.270796
    0.025974        0.693282        29.527778       0.025974        1.011753
            29.527778       16080   33990   1.761622        2.759934
    3.270796        0.037422        0.609866        14.071730       0.037422
            1.254887        14.071730       13076   37008   0.475703
    1.591004        3.181470        0.026418        0.691146
    25.039216       0.026418        0.961103        25.039216       13110
    37086   0.479675        1.590604        3.152124        0.026316
    0.695217        21.490196       0.026316        0.963833
    21.490196       10170   39478   0.307540        0.588086        0.923382
            0.025974        0.765286        29.527778       0.025974
    0.712126        29.527778
0.4     52464   147536  0.298784        1.756777        3.349088
    0.026100        0.691147        30.756757       0.026100        1.117399
            30.756757       16062   34040   2.034483        2.759895
    3.349088        0.037190        0.594947        13.844538       0.037190
            1.460926        13.844538       13096   36798   0.484967
    1.587998        3.156608        0.026100        0.696620
    21.653846       0.026100        1.053171        21.653846       13340
    36744   0.482940        1.587622        3.253086        0.026194
    0.689614        22.529412       0.026194        1.048817
    22.529412       9966    39954   0.298784        0.588282        0.889344
            0.027778        0.769477        30.756757       0.027778
    0.696999        30.756757
```

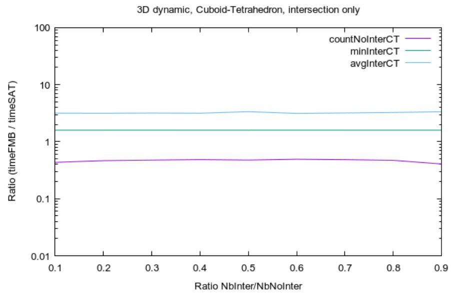
```
0.5     52548   147452  0.261731        1.753130        3.361954
    0.026074        0.677937        29.162162       0.026074        1.215534
            29.162162       16064   33218   2.212185        2.758671
    3.243386        0.037461        0.582163        13.715447       0.037461
            1.670417        13.715447       13120   37332   0.476406
    1.588347        3.361954        0.026214        0.677791
    25.901961       0.026214        1.133069        25.901961       13190
    36672   0.441667        1.591462        3.112623        0.026074
    0.682622        23.040000       0.026074        1.137042
    23.040000       10174   40230   0.261731        0.587549        1.035622
            0.026316        0.752881        29.162162       0.026316
    0.670215        29.162162
0.6     52136   147864  0.297690        1.752979        3.461672
    0.025895        0.687321        29.194444       0.025895        1.326715
            29.194444       15904   34230   1.967329        2.759280
    3.461672        0.037657        0.602129        13.831276       0.037657
            1.896420        13.831276       13122   37162   0.492318
    1.591625        3.135982        0.025895        0.688886
    22.220000       0.025895        1.230530        22.220000       13046
    37022   0.441662        1.587810        3.241248        0.025934
    0.673437        24.420000       0.025934        1.222061
    24.420000       10064   39450   0.297690        0.587225        0.945034
            0.026316        0.772794        29.194444       0.026316
    0.661453        29.194444
0.7     52606   147394  0.304502        1.763538        3.424645
    0.026174        0.692122        29.000000       0.026174        1.442113
            29.000000       16466   33658   2.186667        2.758810
    3.424645        0.036697        0.601924        13.737705       0.036697
            2.111744        13.737705       12932   37112   0.484252
    1.589151        3.193374        0.026625        0.685531
    23.306122       0.026625        1.318065        23.306122       13106
    37078   0.462850        1.590831        3.163997        0.026174
    0.700987        21.480769       0.026174        1.323878
    21.480769       10102   39546   0.304502        0.588575        0.911724
            0.026255        0.766763        29.000000       0.026255
    0.642032        29.000000
0.8     52288   147712  0.305882        1.754773        3.478836
    0.025279        0.689685        30.540541       0.025279        1.541755
            30.540541       16038   34366   1.914320        2.759470
    3.478836        0.037634        0.604981        14.119149       0.037634
            2.328572        14.119149       12896   36720   0.471064
    1.588283        3.258187        0.026856        0.686767
    23.745098       0.026856        1.407979        23.745098       13246
    36790   0.490528        1.590958        3.160247        0.026616
    0.692866        23.490196       0.026616        1.411339
    23.490196       10108   39836   0.305882        0.587738        0.902539
            0.025279        0.762510        30.540541       0.025279
    0.622693        30.540541
0.9     52364   147636  0.293860        1.745450        3.674936
    0.026316        0.683076        31.783784       0.026316        1.639212
            31.783784       15856   34192   1.933252        2.760060
    3.674936        0.037344        0.589482        14.066116       0.037344
            2.543002        14.066116       12962   37346   0.406656
    1.588794        3.355183        0.026604        0.687006
    23.940000       0.026604        1.498615        23.940000       13142
    36504   0.358188        1.590008        3.441595        0.026316
    0.683885        26.714286       0.026316        1.499396
    26.714286       10404   39594   0.293860        0.590674        1.032742
            0.026357        0.759449        31.783784       0.026357
    0.607552        31.783784
```

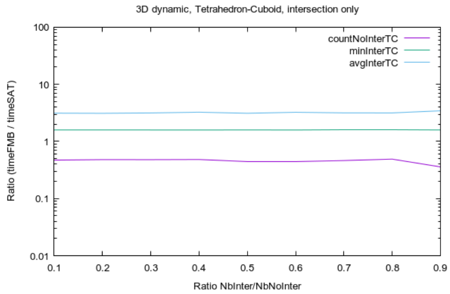

3D dynamic, Total
maxNoInter
minTotal
avgTotal
Ratio (timeFMB / timeSAT)
100
10
1
0.1
0.01
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ratio NbInter/NbNoInter

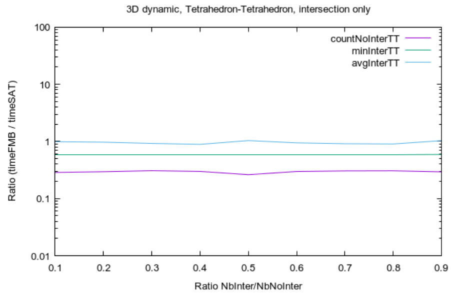

3D dynamic, Total, intersection only
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter
countNoInter
minInter
avgInter
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9

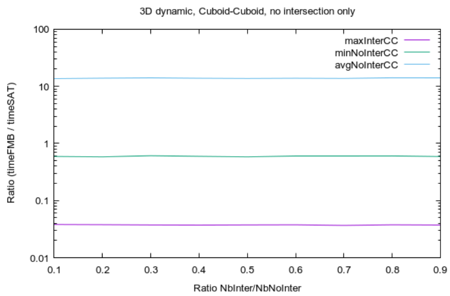

3D dynamic, Total, no intersection only
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter
maxInter
minNoInter
avgNoInter
100
10
1
0.1
0.01
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9

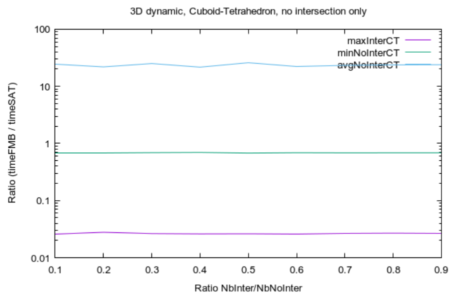

3D dynamic, Cuboid-Cuboid, intersection only
countNoInterCC
minInterCC
avgInterCC
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter

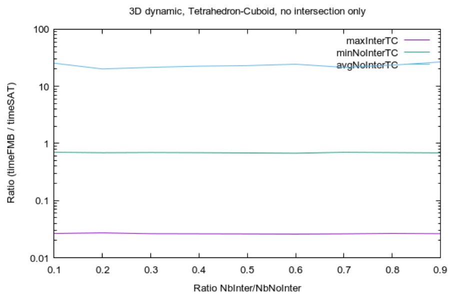

3D dynamic, Cuboid-Tetrahedron, intersection only
countNoInterCT
minInterCT
avgInterCT
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter

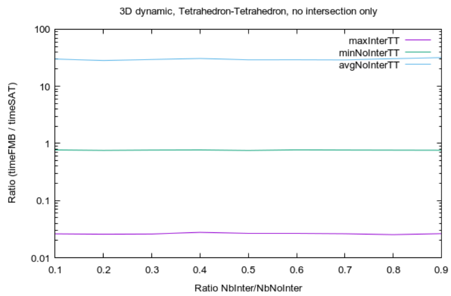

3D dynamic, Tetrahedron-Cuboid, intersection only
countNoInterTC
minInterTC
avgInterTC
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter


3D dynamic, Tetrahedron-Tetrahedron, intersection only
countNoInterTT
minInterTT
avgInterTT
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter


3D dynamic, Cuboid-Cuboid, no intersection only
maxInterCC
minNoInterCC
avgNoInterCC
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter


3D dynamic, Cuboid-Tetrahedron, no intersection only
maxInterCT
minNoInterCT
avgNoInterCT
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter


3D dynamic, Tetrahedron-Cuboid, no intersection only
maxInterTC
minNoInterTC
avgNoInterTC
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter


3D dynamic, Tetrahedron-Tetrahedron, no intersection only
maxInterTT
minNoInterTT
avgNoInterTT
Ratio (timeFMB / timeSAT)
Ratio NbInter/NbNoInter

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.