

The FMB Algorithm

P. Baillehache

January 18, 2020

Abstract

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

Contents

1	The problem as a system of linear inequations	4
1.1	Notations and definitions	4
1.2	Static case	4
1.3	Dynamic case	7
2	Resolution of the problem by Fourier-Motzkin method	11
2.1	The Fourier-Motzkin elimination method	11
2.2	Application of the Fourier-Motzkin method to the intersection problem	13
2.3	About the size of the system of linear inequations	13
3	Algorithms of the solution	15
3.1	2D static	15
3.2	3D static	21
3.3	2D dynamic	29
3.4	3D dynamic	36
4	Implementation of the algorithms in C	44
4.1	Frames	45
4.1.1	Header	45
4.1.2	Body	48
4.2	FMB	69
4.2.1	2D static	69
4.2.2	3D static	78
4.2.3	2D dynamic	89
4.2.4	3D dynamic	99
5	Minimal example of use	110
5.1	2D static	110
5.2	3D static	112
5.3	2D dynamic	114
5.4	3D dynamic	115
6	Unit tests	117
6.1	Code	117
6.1.1	2D static	117
6.1.2	3D static	129
6.1.3	2D dynamic	137
6.1.4	3D dynamic	143
6.2	Results	148

6.2.1	2D static	148
6.2.2	3D static	152
6.2.3	2D dynamic	155
6.2.4	3D dynamic	157
7	Validation against SAT	159
7.1	Code	159
7.1.1	2D static	159
7.1.2	3D static	163
7.1.3	2D dynamic	166
7.1.4	3D dynamic	170
7.2	Results	173
7.2.1	Failures	173
7.2.2	2D static	174
7.2.3	2D dynamic	174
7.2.4	3D static	174
7.2.5	3D dynamic	174
8	Qualification against SAT	174
8.1	Code	175
8.1.1	2D static	175
8.1.2	3D static	186
8.1.3	2D dynamic	197
8.1.4	3D dynamic	209
8.2	Results	220
8.2.1	2D static	221
8.2.2	3D static	226
8.2.3	2D dynamic	231
8.2.4	3D dynamic	236
9	Conclusion	241
10	Annex	241
10.1	Runtime environment	241
10.2	SAT implementation	243
10.2.1	Header	243
10.2.2	Body	244
10.3	Makefile	264
10.3.1	2D static	267
10.3.2	3D static	267
10.3.3	2D dynamic	268

10.3.4	3D dynamic	269
10.3.5	Doc	270
10.4	Dynamic analysis	270

Introduction

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

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

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

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

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

1 The problem as a system of linear inequations

1.1 Notations and definitions

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

1.2 Static case

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

Each vector is of dimension equal to D .

Let's call \mathbb{A} and \mathbb{B} the two Frames tested for intersection. If \mathbb{A} and \mathbb{B} are two cuboids:

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

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

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

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

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

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

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

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

If \mathbb{B} is a tetrahedron:

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

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

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

and for a tetrahedron:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3 Dynamic case

If the frames \mathbb{A} and \mathbb{B} are moving linearly along the vectors $\vec{V}_{\mathbb{A}}$ and $\vec{V}_{\mathbb{B}}$ respectively during the interval of time $t \in [0.0, 1.0]$, the above definition of

the problem is modified as follow.

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

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

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

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

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

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

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

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

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

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

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

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

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

and for a tetrahedron:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2 Resolution of the problem by Fourier-Motzkin method

2.1 The Fourier-Motzkin elimination method

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

for each variable can be obtained if it exists. The variable elimination is performed as follow.

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

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

with

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

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

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

where

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

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

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

and

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

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

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

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

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

The Fourier-Motzkin method can be directly applied to the inequality systems of the previous section, to obtain the bounding box of the intersection, if the system has a solution. If the system has no solution, the method will eventually reach an inconsistent inequality, meaning there is no intersection between the two Frames.

One coordinate \vec{S} , or (\vec{S}, t) in dynamic case, within the bounds obtained by the resolution of the system is expressed in the Frame \mathbb{B} 's coordinates system. One can get the equivalent coordinates \vec{S}' , or (\vec{S}', t) , in the real world's coordinates system as follow:

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

$$(\vec{S}', t) = \left(\vec{O}_{\mathbb{B}} + C_{\mathbb{B}} \cdot \vec{S} + \vec{V}_{\mathbb{B}} \cdot t, t \right) \quad (40)$$

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

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

$$Y < \sum_{i \in I^-} a_i \quad (41)$$

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

2.3 About the size of the system of linear inequations

During implementation in languages where the developer needs to manage memory itself the size of the systems (35) resulting from variable elimination

is necessary but cannot be forecasted. Instead, a maximum size can be calculated as follow.

Let's call n_- , n_+ and n_0 , each in $[0, \mathbb{N}]$, the size of, respectively, \mathcal{I}_- , \mathcal{I}_+ and \mathcal{I}_0 , and N the number of inequalities in the original system and N' the number inequalities in the resulting system. We have:

$$n_- + n_+ + n_0 = N \quad (42)$$

and

$$n_-.n_+ + n_0 = N' \quad (43)$$

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

$$n_- + n_+ = K \quad (44)$$

then,

$$n_-.n_+ = n_-.(K - n_-) \quad (45)$$

then,

$$n_-.n_+ = K.n_- - n_-^2 \quad (46)$$

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

$$n_-.n_+ \leq K^2/2 - K^2/4 \quad (47)$$

or,

$$n_-.n_+ \leq K^2/4 \quad (48)$$

and putting back the definition of K

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

which is also

$$n_-.n_+ \leq N^2/4 \quad (50)$$

From (43) we get,

$$N' \leq N^2/4 + n_0 \quad (51)$$

and finally,

$$N' \leq N^2/4 + N \quad (52)$$

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

	N	N'	N''	N'''
<i>2Dstatic</i>	8	24		
<i>2Ddynamic</i>	10	35	342	
<i>3Dstatic</i>	12	48	624	
<i>3Ddynamic</i>	14	63	1056	279840

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

	N	N'	N''	N'''
<i>2Dstatic</i>	8	11		
<i>2Ddynamic</i>	10	13	21	
<i>3Dstatic</i>	12	20	55	
<i>3Ddynamic</i>	14	22	57	560

3 Algorithms of the solution

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

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

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

3.1 2D static

```

ENUM FrameType
    FrameCuboid,
    FrameTetrahedron
END ENUM

STRUCT AAB2D
    // x,y
    real min[2]

```

```

    real max[2]
END STRUCT

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

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

FUNCTION Frame2DPrint(that)
    IF that.type == FrameTetrahedron
        PRINT "T"
    ELSE IF that.type == FrameCuboid
        PRINT "C"
    END IF
    PRINT "o("
    FOR i = 0 TO 1
        PRINT that.orig[i]
        IF i < 1
            PRINT ","
        END IF
    END FOR
    comp = ["x","y"]
    FOR j = 0 TO 1
        PRINT ") ", comp[j], "("
        FOR i = 0 TO 1
            PRINT that.comp[j][i]
            IF i < 1
                PRINT ","
            END IF
        END FOR
    END FOR
    PRINT ")"
END FUNCTION

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

```



```

END FUNCTION

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

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

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

```

```

FUNCTION Frame2DCreateStatic(type, orig, comp)
    that.type = type
    FOR iAxis = 0 TO 1
        that.orig[iAxis] = orig[iAxis]
        FOR iComp = 0 TO 1
            that.comp[iComp][iAxis] = comp[iComp][iAxis]
        END FOR
    END FOR
    FOR iAxis = 0 TO 1
        min = orig[iAxis]
        max = orig[iAxis]
        FOR iComp = 0 TO 1
            IF that.type == FrameCuboid
                IF that.comp[iComp][iAxis] < 0.0
                    min = min + that.comp[iComp][iAxis]
                END IF
                IF that.comp[iComp][iAxis] > 0.0
                    max = max + that.comp[iComp][iAxis]
                END IF
            ELSE IF that.type == FrameTetrahedron
                IF that.comp[iComp][iAxis] < 0.0 AND
                    min > orig[iAxis] + that.comp[iComp][iAxis]
                    min = orig[iAxis] + that.comp[iComp][iAxis]
                END IF
                IF that.comp[iComp][iAxis] > 0.0 AND
                    max < orig[iAxis] + that.comp[iComp][iAxis]
                    max = orig[iAxis] + that.comp[iComp][iAxis]
                END IF
            END IF
        END FOR
        that.bdgBox.min[iAxis] = min
        that.bdgBox.max[iAxis] = max
    END FOR
    Frame2DUpdateInv(that)
    RETURN that
END FUNCTION

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

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

FST_VAR = 0
SND_VAR = 1

```

```

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

FUNCTION GetBoundLastVar2D(iVar, M, Y, nbRows, bdgBox)
    bdgBox.min[iVar] = 0.0
    bdgBox.max[iVar] = 1.0
    FOR jRow = 0 TO (nbRows - 1)
        IF M[jRow][0] > 0.0
            y = Y[jRow] / M[jRow][0]
            IF bdgBox.max[iVar] > y
                bdgBox.max[iVar] = y
            END IF
        ELSE IF M[jRow][0] < 0.0
            y = Y[jRow] / M[jRow][0]
            IF bdgBox.min[iVar] < y
                bdgBox.min[iVar] = y
            END IF
        END IF
    END FOR
END FUNCTION

FUNCTION GetBoundVar2D(iVar, M, Y, nbRows, nbCols, bdgBox)
    bdgBox.min[iVar] = 0.0
    bdgBox.max[iVar] = 1.0

```

```

FOR iRow = 0 .. TO (nbRows - 1)
  IF M[iRow][0] <> 0.0
    min = -1.0 * Y[iRow]
    max = Y[iRow]
    FOR iCol = 1 .. TO (nbCols - 1)
      IF M[iRow][iCol] > 0.0
        min = min + M[iRow][iCol] * bdgBox.min[iCol + iVar]
        max = max - M[iRow][iCol] * bdgBox.min[iCol + iVar]
      ELSE IF M[iRow][iCol] < 0.0
        min = min + M[iRow][iCol] * bdgBox.max[iCol + iVar]
        max = max - M[iRow][iCol] * bdgBox.max[iCol + iVar]
      END IF
    END FOR
    min = min / (-1.0 * M[iRow][0])
    max = max / M[iRow][0]
    IF bdgBox.min[iVar] > min
      bdgBox.min[iVar] = min
    END IF
    IF bdgBox.max[iVar] < max
      bdgBox.max[iVar] = max
    END IF
  END IF
END FOR
END FUNCTION

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

```

```

        nbRows = nbRows + 1
    END IF
    IF tho.type == FrameCuboid
        M[nbRows][0] = 1.0
        M[nbRows][1] = 0.0
        Y[nbRows] = 1.0
        nbRows = nbRows + 1
        M[nbRows][0] = 0.0
        M[nbRows][1] = 1.0
        Y[nbRows] = 1.0
        nbRows = nbRows + 1
    ELSE
        M[nbRows][0] = 1.0
        M[nbRows][1] = 1.0
        Y[nbRows] = 1.0
        nbRows = nbRows + 1
    END IF
    M[nbRows][0] = -1.0
    M[nbRows][1] = 0.0
    Y[nbRows] = 0.0
    nbRows = nbRows + 1
    M[nbRows][0] = 0.0
    M[nbRows][1] = -1.0
    Y[nbRows] = 0.0
    nbRows = nbRows + 1
    inconsistency = ElimVar2D(M, Y, nbRows, 2, Mp, Yp, nbRowsP)
    IF inconsistency == TRUE
        RETURN FALSE
    END IF
    GetBoundLastVar2D(SND_VAR, Mp, Yp, nbRowsP, bdgBoxLocal)
    IF bdgBoxLocal.min[SND_VAR] >= bdgBoxLocal.max[SND_VAR]
        RETURN FALSE
    ELSE
        GetBoundVar2D(FST_VAR, M, Y, nbRows, 2, bdgBoxLocal)
        bdgBox = bdgBoxLocal
    END IF
    RETURN TRUE
END FUNCTION

origP2D = [0.0, 0.0]
compP2D = [
    [1.0, 0.0],
    [0.0, 1.0]]
P2D = Frame2DCreateStatic(FrameCuboid, origP2D, compP2D)
origQ2D = [0.0, 0.0]
compQ2D = [
    [1.0, 0.0],
    [0.0, 1.0]]
Q2D = Frame2DCreateStatic(FrameCuboid, origQ2D, compQ2D)
isIntersecting2D = FMBTestIntersection2D(P2D, Q2D, bdgBox2DLocal)
IF isIntersecting2D == TRUE
    PRINT "Intersection detected."
    Frame2DExportBdgBox(Q2D, bdgBox2DLocal, bdgBox2D);
    AAB2DPrint(bdgBox2D)
ELSE
    PRINT "No intersection."
END IF

```

3.2 3D static

```

ENUM FrameType

```

```

    FrameCuboid,
    FrameTetrahedron
END ENUM

STRUCT AAB3D
    // x,y,z
    real min[3]
    real max[3]
END STRUCT

STRUCT Frame3D
    FrameType type
    real orig[3]
    // comp[iComp][iAxis]
    real comp[3][3]
    AAB3D bdgBox
    real invComp[3][3]
END STRUCT

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

FUNCTION Frame3DPrint(that)
    IF that.type == FrameTetrahedron
        PRINT "T"
    ELSE IF that.type == FrameCuboid
        PRINT "C"
    END IF
    PRINT "o("
    FOR i = 0 TO 2
        PRINT that.orig[i]
        IF i < 2
            PRINT ", "
        END IF
    END FOR
    comp = ["x", "y", "z"]
    FOR j = 0 TO 2
        PRINT ") ", comp[j], "("
        FOR i = 0 TO 2
            PRINT that.comp[j][i]
            IF i < 2
                PRINT ", "
            END IF
        END FOR
    END FOR
    PRINT ")"
END FUNCTION

FUNCTION AAB3DPrint(that)
    PRINT "minXYZ("
    FOR i = 0 TO 2
        PRINT that.min[i]
        IF i < 2
            PRINT ", "
        END IF
    END FOR
    PRINT ") -maxXYZ("

```

```

FOR i = 0 TO 2
  PRINT that.max[i]
  IF i < 2
    PRINT ","
  END IF
END FOR
PRINT ")"
END FUNCTION

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

FUNCTION Frame3DImportFrame(P, Q, Qp)
FOR i = 0 TO 2
  v[i] = Q.orig[i] - P.orig[i]
END FOR
FOR i = 0 TO 2
  Qp.orig[i] = 0.0
  FOR j = 0 TO 2
    Qp.orig[i] = Qp.orig[i] + P.invComp[j][i] * v[j]
    Qp.comp[j][i] = 0.0
    FOR k = 0 TO 2
      Qp.comp[j][i] = Qp.comp[j][i] + P.invComp[k][i] * Q.comp[j][k]
    END FOR
  END FOR
END FOR
END FUNCTION

FUNCTION Frame3DUpdateInv(that)
  det =

```

```

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

FUNCTION Frame3DCreateStatic(type, orig, comp)
    that.type = type
    FOR iAxis = 0 TO 2
        that.orig[iAxis] = orig[iAxis]
        FOR iComp = 0 TO 2
            that.comp[iComp][iAxis] = comp[iComp][iAxis]
        END FOR
    END FOR
    FOR iAxis = 0 TO 2
        min = orig[iAxis]
        max = orig[iAxis]
        FOR iComp = 0 TO 2
            IF that.type == FrameCuboid
                IF that.comp[iComp][iAxis] < 0.0
                    min = min + that.comp[iComp][iAxis]
                END IF
                IF that.comp[iComp][iAxis] > 0.0
                    max = max + that.comp[iComp][iAxis]
                END IF
            ELSE IF that.type == FrameTetrahedron
                IF that.comp[iComp][iAxis] < 0.0 AND
                    min > orig[iAxis] + that.comp[iComp][iAxis]
                    min = orig[iAxis] + that.comp[iComp][iAxis]
                END IF
                IF that.comp[iComp][iAxis] > 0.0 AND
                    max < orig[iAxis] + that.comp[iComp][iAxis]
                    max = orig[iAxis] + that.comp[iComp][iAxis]
                END IF
            END IF
        END FOR
        that.bdgBox.min[iAxis] = min
        that.bdgBox.max[iAxis] = max
    END FOR
    Frame3DUpdateInv(that)
    RETURN that
END FUNCTION

```



```

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

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

FST_VAR = 0
SND_VAR = 1
THD_VAR = 2

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

```

```

        nbRemainRows = nbRemainRows + 1
    END IF
END FOR
RETURN FALSE
END FUNCTION

FUNCTION GetBoundLastVar3D(iVar, M, Y, nbRows, bdgBox)
    bdgBox.min[iVar] = 0.0
    bdgBox.max[iVar] = 1.0
    FOR jRow = 0 TO (nbRows - 1)
        IF M[jRow][0] > 0.0
            y = Y[jRow] / M[jRow][0]
            IF bdgBox.max[iVar] > y
                bdgBox.max[iVar] = y
            END IF
        ELSE IF M[jRow][0] < 0.0
            y = Y[jRow] / M[jRow][0]
            IF bdgBox.min[iVar] < y
                bdgBox.min[iVar] = y
            END IF
        END IF
    END FOR
END FUNCTION

FUNCTION GetBoundVar3D(iVar, M, Y, nbRows, nbCols, bdgBox)
    bdgBox.min[iVar] = 0.0
    bdgBox.max[iVar] = 1.0
    FOR iRow = 0 .. TO (nbRows - 1)
        IF M[iRow][0] <> 0.0
            min = -1.0 * Y[iRow]
            max = Y[iRow]
            FOR iCol = 1 .. TO (nbCols - 1)
                IF M[iRow][iCol] > 0.0
                    min = min + M[iRow][iCol] * bdgBox.min[iCol + iVar]
                    max = max - M[iRow][iCol] * bdgBox.min[iCol + iVar]
                ELSE IF M[iRow][iCol] < 0.0
                    min = min + M[iRow][iCol] * bdgBox.max[iCol + iVar]
                    max = max - M[iRow][iCol] * bdgBox.max[iCol + iVar]
                END IF
            END FOR
            min = min / (-1.0 * M[iRow][0])
            max = max / M[iRow][0]
            IF bdgBox.min[iVar] > min
                bdgBox.min[iVar] = min
            END IF
            IF bdgBox.max[iVar] < max
                bdgBox.max[iVar] = max
            END IF
        END IF
    END FOR
END FUNCTION

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

```

```

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

```

```

        nbRows = nbRows + 1
        M[nbRows][0] = 0.0
        M[nbRows][1] = 1.0
        M[nbRows][2] = 0.0
        Y[nbRows] = 1.0
        nbRows = nbRows + 1
        M[nbRows][0] = 0.0
        M[nbRows][1] = 0.0
        M[nbRows][2] = 1.0
        Y[nbRows] = 1.0
        nbRows = nbRows + 1
    ELSE
        M[nbRows][0] = 1.0
        M[nbRows][1] = 1.0
        M[nbRows][2] = 1.0
        Y[nbRows] = 1.0
        nbRows = nbRows + 1
    END IF
    M[nbRows][0] = -1.0
    M[nbRows][1] = 0.0
    M[nbRows][2] = 0.0
    Y[nbRows] = 0.0
    nbRows = nbRows + 1
    M[nbRows][0] = 0.0
    M[nbRows][1] = -1.0
    M[nbRows][2] = 0.0
    Y[nbRows] = 0.0
    nbRows = nbRows + 1
    M[nbRows][0] = 0.0
    M[nbRows][1] = 0.0
    M[nbRows][2] = -1.0
    Y[nbRows] = 0.0
    nbRows = nbRows + 1
    inconsistency =
        ElimVar3D(M, Y, nbRows, 3, Mp, Yp, nbRowsP)
    IF inconsistency == TRUE
        RETURN FALSE
    END IF
    inconsistency =
        ElimVar3D(Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
    IF inconsistency == TRUE
        RETURN FALSE
    END IF
    GetBoundLastVar3D(THD_VAR, Mpp, Ypp, nbRowsPP, bdgBoxLocal)
    IF bdgBoxLocal.min[THD_VAR] >= bdgBoxLocal.max[THD_VAR]
        RETURN FALSE
    ELSE
        GetBoundVar3D(SND_VAR, Mp, Yp, nbRowsP, 2, bdgBoxLocal)
        GetBoundVar3D(FST_VAR, M, Y, nbRows, 3, bdgBoxLocal)
        bdgBox = bdgBoxLocal
    END IF
    RETURN TRUE
END FUNCTION

origP3D = [0.0, 0.0, 0.0]
compP3D = [
    [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0],
    [0.0, 0.0, 1.0]]
P3D = Frame3DCreateStatic(FrameTetrahedron, origP3D, compP3D)
origQ3D = [0.0, 0.0, 0.0]
compQ3D = [

```

```

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

```

3.3 2D dynamic

```

ENUM FrameType
    FrameCuboid,
    FrameTetrahedron
END ENUM

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

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

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

FUNCTION Frame2DTimePrint(that)
    IF that.type == FrameTetrahedron
        PRINT "T"
    ELSE IF that.type == FrameCuboid
        PRINT "C"
    END IF
    PRINT "o("
    FOR i = 0 TO 1
        PRINT that.orig[i]
        IF i < 1
            PRINT ","
        END IF
    END FOR
    PRINT ") s("
    FOR i = 0 TO 1
        PRINT that.speed[i]
        IF i < 1

```

```

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

FUNCTION AAB2DTimePrint(that)
    PRINT "minXYT("
    FOR i = 0 TO 2
        PRINT that.min[i]
        IF i < 2
            PRINT ","
        END IF
    END FOR
    PRINT ")-maxXYT("
    FOR i = 0 TO 2
        PRINT that.max[i]
        IF i < 2
            PRINT ","
        END IF
    END FOR
    PRINT ")"
END FUNCTION

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

```

```

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

FUNCTION Frame2DTimeImportFrame(P, Q, Qp)
    FOR i = 0 TO 1
        v[i] = Q.orig[i] - P.orig[i]
        s[i] = Q.speed[i] - P.speed[i]
    END FOR
    FOR i = 0 TO 1
        Qp.orig[i] = 0.0
        Qp.speed[i] = 0.0
        FOR j = 0 TO 1
            Qp.orig[i] = Qp.orig[i] + P.invComp[j][i] * v[j]
            Qp.speed[i] = Qp.speed[i] + P.invComp[j][i] * s[j]
            Qp.comp[j][i] = 0.0
            FOR k = 0 TO 1
                Qp.comp[j][i] = Qp.comp[j][i] + P.invComp[k][i] * Q.comp[j][k]
            END FOR
        END FOR
    END FOR
END FUNCTION

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

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

```

```

        END IF
    ELSE IF that.type == FrameTetrahedron
        IF that.comp[iComp][iAxis] < 0.0 AND
            min > orig[iAxis] + that.comp[iComp][iAxis]
            min = orig[iAxis] + that.comp[iComp][iAxis]
        END IF
        IF that.comp[iComp][iAxis] > 0.0 AND
            max < orig[iAxis] + that.comp[iComp][iAxis]
            max = orig[iAxis] + that.comp[iComp][iAxis]
        END IF
    END IF
END FOR
IF that.speed[iAxis] < 0.0
    min = min + that.speed[iAxis]
END IF
IF that.speed[iAxis] > 0.0
    max = max + that.speed[iAxis]
END IF
that.bdgBox.min[iAxis] = min
that.bdgBox.max[iAxis] = max
END FOR
that.bdgBox.min[2] = 0.0
that.bdgBox.max[2] = 1.0
Frame2DTimeUpdateInv(that)
RETURN that
END FUNCTION

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

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

FST_VAR = 0
SND_VAR = 1
THD_VAR = 2

FUNCTION ElimVar2DTime(M, Y, nbRows, nbCols, Mp, Yp, nbRemainRows)
    nbRemainRows = 0
    FOR iRow = 0 TO (nbRows - 2)
        IF M[iRow][0] <> 0.0
            FOR jRow = (iRow + 1) TO (nbRows - 1)
                IF sgn(M[iRow][0]) <> sgn(M[jRow][0]) AND
                    M[jRow][0] <> 0.0
                    sumNegCoeff = 0.0
                    jCol = 0
                END IF
            END FOR
        END IF
    END FOR
    RETURN nbRemainRows
END FUNCTION

```



```

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

FUNCTION GetBoundLastVar2DTime(iVar, M, Y, nbRows, bdgBox)
    bdgBox.min[iVar] = 0.0
    bdgBox.max[iVar] = 1.0
    FOR jRow = 0 TO (nbRows - 1)
        IF M[jRow][0] > 0.0
            y = Y[jRow] / M[jRow][0]
            IF bdgBox.max[iVar] > y
                bdgBox.max[iVar] = y
            END IF
        ELSE IF M[jRow][0] < 0.0
            y = Y[jRow] / M[jRow][0]
            IF bdgBox.min[iVar] < y
                bdgBox.min[iVar] = y
            END IF
        END IF
    END FOR
END FUNCTION

FUNCTION GetBoundVar2DTime(iVar, M, Y, nbRows, nbCols, bdgBox)
    bdgBox.min[iVar] = 0.0
    bdgBox.max[iVar] = 1.0
    FOR iRow = 0 .. TO (nbRows - 1)
        IF M[iRow][0] <> 0.0
            min = -1.0 * Y[iRow]
            max = Y[iRow]
            FOR iCol = 1 .. TO (nbCols - 1)
                IF M[iRow][iCol] > 0.0
                    min = min + M[iRow][iCol] * bdgBox.min[iCol + iVar]
                    max = max - M[iRow][iCol] * bdgBox.min[iCol + iVar]
                ELSE IF M[iRow][iCol] < 0.0

```

```

        min = min + M[iRow][iCol] * bdgBox.max[iCol + iVar]
        max = max - M[iRow][iCol] * bdgBox.max[iCol + iVar]
    END IF
END FOR
min = min / (-1.0 * M[iRow][0])
max = max / M[iRow][0]
IF bdgBox.min[iVar] > min
    bdgBox.min[iVar] = min
END IF
IF bdgBox.max[iVar] < max
    bdgBox.max[iVar] = max
END IF
END IF
END FOR
END FUNCTION

FUNCTION FMBTestIntersection2DTime(that, tho, bdgBox)
    Frame2DTimeImportFrame(that, tho, thoProj)
    M[0][0] = -thoProj.comp[0][0]
    M[0][1] = -thoProj.comp[1][0]
    M[0][2] = -thoProj.speed[0]
    Y[0] = thoProj.orig[0]
    IF (Y[0] < neg(M[0][0]) + neg(M[0][1]) + neg(M[0][2]))
        RETURN FALSE
    END IF
    M[1][0] = -thoProj.comp[0][1]
    M[1][1] = -thoProj.comp[1][1]
    M[1][2] = -thoProj.speed[1]
    Y[1] = thoProj.orig[1]
    IF (Y[1] < neg(M[1][0]) + neg(M[1][1]) + neg(M[1][2]))
        RETURN FALSE
    END IF
    nbRows = 2
    IF that.type == FrameCuboid
        M[nbRows][0] = thoProj.comp[0][0]
        M[nbRows][1] = thoProj.comp[1][0]
        M[nbRows][2] = thoProj.speed[0]
        Y[nbRows] = 1.0 - thoProj.orig[0]
        IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
            neg(M[nbRows][2])
            RETURN FALSE
        END IF
        nbRows = nbRows + 1
        M[nbRows][0] = thoProj.comp[0][1]
        M[nbRows][1] = thoProj.comp[1][1]
        M[nbRows][2] = thoProj.speed[1]
        Y[nbRows] = 1.0 - thoProj.orig[1]
        IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
            neg(M[nbRows][2])
            RETURN FALSE
        END IF
        nbRows = nbRows + 1
    ELSE
        M[nbRows][0] = thoProj.comp[0][0] + thoProj.comp[0][1]
        M[nbRows][1] = thoProj.comp[1][0] + thoProj.comp[1][1]
        M[nbRows][2] = thoProj.speed[0] + thoProj.speed[1]
        Y[nbRows] = 1.0 - thoProj.orig[0] - thoProj.orig[1]
        IF Y[nbRows] < neg(M[nbRows][0]) + neg(M[nbRows][1]) +
            neg(M[nbRows][2])
            RETURN FALSE
        END IF
        nbRows = nbRows + 1
    END IF
END FUNCTION

```

```

END IF
IF tho.type == FrameCuboid
    M[nbRows][0] = 1.0
    M[nbRows][1] = 0.0
    M[nbRows][2] = 0.0
    Y[nbRows] = 1.0
    nbRows = nbRows + 1
    M[nbRows][0] = 0.0
    M[nbRows][1] = 1.0
    M[nbRows][2] = 0.0
    Y[nbRows] = 1.0
    nbRows = nbRows + 1
ELSE
    M[nbRows][0] = 1.0
    M[nbRows][1] = 1.0
    M[nbRows][2] = 0.0
    Y[nbRows] = 1.0
    nbRows = nbRows + 1
END IF
M[nbRows][0] = -1.0
M[nbRows][1] = 0.0
M[nbRows][2] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = -1.0
M[nbRows][2] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = 1.0
Y[nbRows] = 1.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = -1.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
inconsistency =
    ElimVar2DTime(M, Y, nbRows, 3, Mp, Yp, nbRowsP)
IF inconsistency == TRUE
    RETURN FALSE
END IF
inconsistency =
    ElimVar2DTime(Mp, Yp, nbRowsP, 2, Mpp, Ypp, nbRowsPP)
IF inconsistency == TRUE
    RETURN FALSE
END IF
GetBoundLastVar2DTime(THD_VAR, Mpp, Ypp, nbRowsPP, bdgBoxLocal)
IF bdgBoxLocal.min[THD_VAR] >= bdgBoxLocal.max[THD_VAR]
    RETURN FALSE
ELSE
    GetBoundVar2DTime(SND_VAR, Mp, Yp, nbRowsP, 2, bdgBoxLocal)
    GetBoundVar2DTime(FST_VAR, M, Y, nbRows, 3, bdgBoxLocal)
    bdgBox = bdgBoxLocal
END IF
RETURN TRUE
END FUNCTION

origP2DTime = [0.0, 0.0]
speedP2DTime = [0.0, 0.0]

```

```

compP2DTime = [
    [1.0, 0.0],
    [0.0, 1.0]]
P2DTime =
    Frame2DTimeCreateStatic(
        FrameCuboid, origP2DTime, speedP2DTime, compP2DTime)
origQ2DTime = [0.0,0.0]
speedQ2DTime = [0.0,0.0]
compQ2DTime = [
    [1.0, 0.0],
    [0.0, 1.0]]
Q2DTime =
    Frame2DTimeCreateStatic(
        FrameCuboid, origQ2DTime, speedQ2DTime, compQ2DTime)
isIntersecting2DTime =
    FMBTestIntersection2DTime(P2DTime, Q2DTime, bdgBox2DTimeLocal)
IF isIntersecting2DTime == TRUE
    PRINT "Intersection detected."
    Frame2DTimeExportBdgBox(Q2DTime, bdgBox2DTimeLocal, bdgBox2DTime)
    AABBB2DTimePrint(bdgBox2DTime)
ELSE
    PRINT "No intersection."
END IF

```

3.4 3D dynamic

```

ENUM FrameType
    FrameCuboid,
    FrameTetrahedron
END ENUM

STRUCT AABBB3DTime
    // x,y,z,t
    real min[4]
    real max[4]
END STRUCT

STRUCT Frame3DTime
    FrameType type
    real orig[3]
    // comp[iComp][iAxis]
    real comp[3][3]
    AABBB3DTime bdgBox
    real invComp[3][3]
    real speed[3]
END STRUCT

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

FUNCTION Frame3DTimePrint(that)
    IF that.type == FrameTetrahedron
        PRINT "T"
    ELSE IF that.type == FrameCuboid
        PRINT "C"
    END IF

```

```

PRINT "o("
FOR i = 0 TO 2
  PRINT that.orig[i]
  IF i < 2
    PRINT ","
  END IF
END FOR
PRINT " s("
FOR i = 0 TO 2
  PRINT that.speed[i]
  IF i < 2
    PRINT ","
  END IF
END FOR
comp = ["x", "y", "z"]
FOR j = 0 TO 2
  PRINT " ", comp[j], "("
  FOR i = 0 TO 2
    PRINT that.comp[j][i]
    IF i < 2
      PRINT ","
    END IF
  END FOR
END FOR
PRINT ")"
END FUNCTION

FUNCTION AAB3DTimePrint(that)
  PRINT "minXYZT("
  FOR i = 0 TO 3
    PRINT that.min[i]
    IF i < 3
      PRINT ","
    END IF
  END FOR
  PRINT ") -maxXYZT("
  FOR i = 0 TO 3
    PRINT that.max[i]
    IF i < 3
      PRINT ","
    END IF
  END FOR
  PRINT ")"
END FUNCTION

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

```

```

        v[i] = bdgBox.min[i]
    END IF
END FOR
FOR i = 0 TO 2
    w[i] = that.orig[i]
    FOR j = 0 TO 2
        w[i] = w[i] + that.comp[j][i] * v[j]
    END FOR
END FOR
FOR i = 0 TO 2
    IF bdgBoxProj.min[i] > w[i] + that.speed[i] * bdgBox.min[3]
        bdgBoxProj.min[i] = w[i] + that.speed[i] * bdgBox.min[3]
    END IF
    IF bdgBoxProj.min[i] > w[i] + that.speed[i] * bdgBox.max[3]
        bdgBoxProj.min[i] = w[i] + that.speed[i] * bdgBox.max[3]
    END IF
    IF bdgBoxProj.max[i] < w[i] + that.speed[i] * bdgBox.min[3]
        bdgBoxProj.max[i] = w[i] + that.speed[i] * bdgBox.min[3]
    END IF
    IF bdgBoxProj.max[i] < w[i] + that.speed[i] * bdgBox.max[3]
        bdgBoxProj.max[i] = w[i] + that.speed[i] * bdgBox.max[3]
    END IF
END FOR
END FOR
END FUNCTION

FUNCTION Frame3DTimeImportFrame(P, Q, Qp)
    FOR i = 0 TO 2
        v[i] = Q.orig[i] - P.orig[i]
        s[i] = Q.speed[i] - P.speed[i]
    END FOR
    FOR i = 0 TO 2
        Qp.orig[i] = 0.0
        Qp.speed[i] = 0.0
        FOR j = 0 TO 2
            Qp.orig[i] = Qp.orig[i] + P.invComp[j][i] * v[j]
            Qp.speed[i] = Qp.speed[i] + P.invComp[j][i] * s[j]
            Qp.comp[j][i] = 0.0
            FOR k = 0 TO 2
                Qp.comp[j][i] = Qp.comp[j][i] + P.invComp[k][i] * Q.comp[j][k]
            END FOR
        END FOR
    END FOR
END FOR
END FUNCTION

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

```

```

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

```

```

FUNCTION Frame3DTimeCreateStatic(type, orig, comp)
that.type = type
FOR iAxis = 0 TO 2
    that.orig[iAxis] = orig[iAxis]
    that.speed[iAxis] = speed[iAxis]
    FOR iComp = 0 TO 2
        that.comp[iComp][iAxis] = comp[iComp][iAxis]
    END FOR
END FOR
FOR iAxis = 0 TO 2
    min = orig[iAxis]
    max = orig[iAxis]
    FOR iComp = 0 TO 2
        IF that.type == FrameCuboid
            IF that.comp[iComp][iAxis] < 0.0
                min = min + that.comp[iComp][iAxis]
            END IF
            IF that.comp[iComp][iAxis] > 0.0
                max = max + that.comp[iComp][iAxis]
            END IF
        ELSE IF that.type == FrameTetrahedron
            IF that.comp[iComp][iAxis] < 0.0 AND
                min > orig[iAxis] + that.comp[iComp][iAxis]
                min = orig[iAxis] + that.comp[iComp][iAxis]
            END IF
            IF that.comp[iComp][iAxis] > 0.0 AND
                max < orig[iAxis] + that.comp[iComp][iAxis]
                max = orig[iAxis] + that.comp[iComp][iAxis]
            END IF
        END IF
    END FOR
    IF that.speed[iAxis] < 0.0
        min = min + that.speed[iAxis]
    END IF
    IF that.speed[iAxis] > 0.0
        max = max + that.speed[iAxis]
    END IF
    that.bdgBox.min[iAxis] = min
    that.bdgBox.max[iAxis] = max
END FOR
that.bdgBox.min[3] = 0.0
that.bdgBox.max[3] = 1.0
Frame3DTimeUpdateInv(that)
RETURN that
END FUNCTION

```

```

FUNCTION Sgn(v)
IF 0.0 < v
    a = 1

```

```

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

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

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

FUNCTION ElimVar3DTime(M, Y, nbRows, nbCols, Mp, Yp, nbRemainRows)
  nbRemainRows = 0
  FOR iRow = 0 TO (nbRows - 2)
    IF M[iRow][0] <> 0.0
      FOR jRow = (iRow + 1) TO (nbRows - 1)
        IF sgn(M[iRow][0]) <> sgn(M[jRow][0]) AND
           M[jRow][0] <> 0.0
          sumNegCoeff = 0.0
          jCol = 0
          FOR iCol = 1 TO (nbCols - 1)
            Mp[nbRemainRows][jCol] =
              M[iRow][iCol] / ABS(M[iRow][0]) +
              M[jRow][iCol] / ABS(M[jRow][0])
            sumNegCoeff = sumNegCoeff + neg(Mp[nbRemainRows][jCol])
            jCol = jCol + 1
          END FOR
          Yp[nbRemainRows] =
            Y[iRow] / ABS(M[iRow][0]) +
            Y[jRow] / ABS(M[jRow][0])
          IF Yp[nbRemainRows] < sumNegCoeff
            RETURN TRUE
          END IF
          nbRemainRows = nbRemainRows + 1
        END IF
      END FOR
    END IF
  END FOR
  FOR iRow = 0 TO (nbRows - 1)
    IF M[iRow][0] == 0.0
      jCol = 0
      FOR iCol = 1 TO (nbCols - 1)
        Mp[nbRemainRows][jCol] = M[iRow][iCol]
        jCol = jCol + 1
      END FOR
      Yp[nbRemainRows] = Y[iRow]
      nbRemainRows = nbRemainRows + 1
    END IF
  END FOR
END FUNCTION

```



```

    RETURN FALSE
END FUNCTION

FUNCTION GetBoundLastVar3DTime(iVar, M, Y, nbRows, bdgBox)
    bdgBox.min[iVar] = 0.0
    bdgBox.max[iVar] = 1.0
    FOR jRow = 0 TO (nbRows - 1)
        IF M[jRow][0] > 0.0
            y = Y[jRow] / M[jRow][0]
            IF bdgBox.max[iVar] > y
                bdgBox.max[iVar] = y
            END IF
        ELSE IF M[jRow][0] < 0.0
            y = Y[jRow] / M[jRow][0]
            IF bdgBox.min[iVar] < y
                bdgBox.min[iVar] = y
            END IF
        END IF
    END FOR
END FUNCTION

FUNCTION GetBoundVar3DTime(iVar, M, Y, nbRows, nbCols, bdgBox)
    bdgBox.min[iVar] = 0.0
    bdgBox.max[iVar] = 1.0
    FOR iRow = 0 .. TO (nbRows - 1)
        IF M[iRow][0] <> 0.0
            min = -1.0 * Y[iRow]
            max = Y[iRow]
            FOR iCol = 1 .. TO (nbCols - 1)
                IF M[iRow][iCol] > 0.0
                    min = min + M[iRow][iCol] * bdgBox.min[iCol + iVar]
                    max = max - M[iRow][iCol] * bdgBox.min[iCol + iVar]
                ELSE IF M[iRow][iCol] < 0.0
                    min = min + M[iRow][iCol] * bdgBox.max[iCol + iVar]
                    max = max - M[iRow][iCol] * bdgBox.max[iCol + iVar]
                END IF
            END FOR
            min = min / (-1.0 * M[iRow][0])
            max = max / M[iRow][0]
            IF bdgBox.min[iVar] > min
                bdgBox.min[iVar] = min
            END IF
            IF bdgBox.max[iVar] < max
                bdgBox.max[iVar] = max
            END IF
        END IF
    END FOR
END FUNCTION

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

```

```

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

```

```

M[nbRows][1] = 0.0
M[nbRows][2] = 0.0
M[nbRows][3] = 0.0
Y[nbRows] = 1.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 1.0
M[nbRows][2] = 0.0
M[nbRows][3] = 0.0
Y[nbRows] = 1.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = 1.0
M[nbRows][3] = 0.0
Y[nbRows] = 1.0
nbRows = nbRows + 1
ELSE
M[nbRows][0] = 1.0
M[nbRows][1] = 1.0
M[nbRows][2] = 1.0
M[nbRows][3] = 0.0
Y[nbRows] = 1.0
nbRows = nbRows + 1
END IF
M[nbRows][0] = -1.0
M[nbRows][1] = 0.0
M[nbRows][2] = 0.0
M[nbRows][3] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = -1.0
M[nbRows][2] = 0.0
M[nbRows][3] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = -1.0
M[nbRows][3] = 0.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = 0.0
M[nbRows][3] = 1.0
Y[nbRows] = 1.0
nbRows = nbRows + 1
M[nbRows][0] = 0.0
M[nbRows][1] = 0.0
M[nbRows][2] = 0.0
M[nbRows][3] = -1.0
Y[nbRows] = 0.0
nbRows = nbRows + 1
inconsistency =
ElimVar3DTime(M, Y, nbRows, 4, Mp, Yp, nbRowsP)
IF inconsistency == TRUE
RETURN FALSE
END IF
inconsistency =
ElimVar3DTime(Mp, Yp, nbRowsP, 3, Mpp, Ypp, nbRowsPP)

```

```

    IF inconsistency == TRUE
        RETURN FALSE
    END IF
    inconsistency =
        ElimVar3DTime(Mpp, Ypp, nbRowsPP, 2, Mppp, Yppp, nbRowsPPP)
    IF inconsistency == TRUE
        RETURN FALSE
    END IF
    GetBoundLastVar3DTime(FOR_VAR, Mppp, Yppp, nbRowsPPP, bdgBoxLocal)
    IF bdgBoxLocal.min[FOR_VAR] >= bdgBoxLocal.max[FOR_VAR]
        RETURN FALSE
    ELSE
        GetBoundVar3DTime(THD_VAR, Mpp, Ypp, nbRowsPP, 2, bdgBoxLocal)
        GetBoundVar3DTime(SND_VAR, Mp, Yp, nbRowsP, 3, bdgBoxLocal)
        GetBoundVar3DTime(FST_VAR, M, Y, nbRows, 4, bdgBoxLocal)
        bdgBox = bdgBoxLocal
    END IF
    RETURN TRUE
END FUNCTION

origP3DTime = [0.0, 0.0, 0.0]
speedP3DTime = [0.0, 0.0, 0.0]
compP3DTime = [
    [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0],
    [0.0, 0.0, 1.0]]
P3DTime =
    Frame3DTimeCreateStatic(
        FrameCuboid, origP3DTime, speedP3DTime, compP3DTime)
origQ3DTime = [0.0, 0.0, 0.0]
speedQ3DTime = [0.0, 0.0, 0.0]
compQ3DTime = [
    [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0],
    [0.0, 0.0, 1.0]]
Q3DTime =
    Frame3DTimeCreateStatic(
        FrameCuboid, origQ3DTime, speedQ3DTime, compQ3DTime)
isIntersecting3DTime =
    FMBTestIntersection3DTime(P3DTime, Q3DTime, bdgBox3DTimeLocal)
IF isIntersecting3DTime == TRUE
    PRINT "Intersection detected."
    Frame3DTimeExportBdgBox(Q3DTime, bdgBox3DTimeLocal, bdgBox3DTime)
    AAB3DTimePrint(bdgBox3DTime)
ELSE
    PRINT "No intersection."
END IF

```

4 Implementation of the algorithms in C

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

4.1 Frames

4.1.1 Header

```
#ifndef __FRAME_H_
#define __FRAME_H_

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

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

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

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

typedef enum {
    FrameCuboid,
    FrameTetrahedron
} FrameType;

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

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

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

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

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

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

typedef struct {
```

```

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

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

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

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

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

// Print the Frame that on stdout
// Output format is
// T/C  <- type of Frame
// o(orig[0], orig[1], orig[2])
// s(speed[0], speed[1], speed[2])
// x(comp[0][0], comp[0][1], comp[0][2])
// y(comp[1][0], comp[1][1], comp[1][2])
// z(comp[2][0], comp[2][1], comp[2][2])
void Frame2DPrint(const Frame2D* const that);
void Frame3DPrint(const Frame3D* const that);
void Frame2DTimePrint(const Frame2DTime* const that);
void Frame3DTimePrint(const Frame3DTime* const that);

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

```

```

    const FrameType type,
        const double orig[3],
        const double comp[3][3]);
Frame2DTime Frame2DTimeCreateStatic(
    const FrameType type,
        const double orig[2],
        const double speed[2],
        const double comp[2][2]);
Frame3DTime Frame3DTimeCreateStatic(
    const FrameType type,
        const double orig[3],
        const double speed[3],
        const double comp[3][3]);

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

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

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

#endif

```

4.1.2 Body

```
#include "frame.h"

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

#define EPSILON 0.0000001

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

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

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

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

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

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

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

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

        }
    }

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

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

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

            if (that.type == FrameCuboid) {

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

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

                }
            }
        }
    }
}
```



```

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

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

        }

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

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

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

        }

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

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

        }

    }

}

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

}

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

// Return the new Frame
return that;

}

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

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

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

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

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

        }

    }

}

```

```

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

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

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

        if (that.type == FrameCuboid) {

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

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

            }

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

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

            }

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

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

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

            }

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

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

            }

        }

    }

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

}

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

// Return the new Frame
return that;

}

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

```

```

        const double comp[2][2]) {

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

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

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

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

    }

}

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

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

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

        if (that.type == FrameCuboid) {

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

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

            }

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

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

            }

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

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

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

            }

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

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

            }

        }

    }

}

```

```

    }

    if (that.speed[iAxis] < 0.0) {
        min += that.speed[iAxis];
    }

    if (that.speed[iAxis] > 0.0) {
        max += that.speed[iAxis];
    }

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

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

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

// Return the new Frame
return that;
}

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

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

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

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

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

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

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

```

```

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

    if (that.type == FrameCuboid) {

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

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

        }

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

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

        }

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

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

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

        }

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

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

        }

    }

}

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

    min += that.speed[iAxis];

}

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

    max += that.speed[iAxis];

}

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

}

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

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

// Return the new Frame
return that;

```

```

}

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

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

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

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

void Frame3DUpdateInv(Frame3D* const that) {

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

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

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

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

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

    double det = tc[0][0] * tc[1][1] - tc[1][0] * tc[0][1];
    if (fabs(det) < EPSILON) {

```

```

    fprintf(stderr,
        "FrameUpdateInv: det == 0.0\n");
    exit(1);
}

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

void Frame3DTimeUpdateInv(Frame3DTime* const that) {

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

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

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

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

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

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

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

```

```

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

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

        qpo[i] = 0.0;

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

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

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

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

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

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

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

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

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

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

        qpo[i] = 0.0;

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

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

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

```



```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```

        Frame3DTime* const Qp) {

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

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

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

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

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

}

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

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

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

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

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

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

        }

    }

}

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

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

```

```

        double* bbpma = bdgBoxProj->max;

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

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

    bbpma[i] = to[i];

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

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

    }

    bbpmi[i] = bbpma[i];
}

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

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

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

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

    }

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

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

        w[i] = to[i];

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

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

        }

    }

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

```

```

        if (bbpmi[i] > w[i]) {
            bbpma[i] = w[i];
        }
        if (bbpma[i] < w[i]) {
            bbpma[i] = w[i];
        }
    }
}

}

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

    // Shortcuts
    const double* to      = that->orig;
    const double* bbmi    = bdgBox->min;
    const double* bbma    = bdgBox->max;
    double* 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 (bbpma[i] < w[i] + ts[i] * bbma[2]) {
            bbpma[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* bbpma        = bdgBoxProj->min;
    double* bbpma        = bdgBoxProj->max;
    const double (*tc)[3] = that->comp;

    // The time component is not affected
    bbpma[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];

        }

        bbpma[i] = bbpma[i];

    }

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

```

```

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

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

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

}

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

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

    w[i] = to[i];

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

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

    }

}

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

    if (bbpmi[i] > w[i] + 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 AAB2DPrint(const AAB2D* 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 AAB3DPrint(const AAB3D* 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 AAB2DTimePrint(const AAB2DTime* 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 AAB3DTimePrint(const AAB3DTime* const that) {

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

        printf("%f", that->min[i]);
        if (i < 3)
            printf(",");
    }
    printf("-maxXYZT(");
    for (int i = 0;
        i < 4;
        ++i) {

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

// Print the Frame that on stdout
// Output format is
// T/C <- type of Frame
// o(orig[0], orig[1], orig[2])
// s(speed[0], speed[1], speed[2])
// x(comp[0][0], comp[0][1], comp[0][2])
// y(comp[1][0], comp[1][1], comp[1][2])
// z(comp[2][0], comp[2][1], comp[2][2])
void Frame2DPrint(const Frame2D* const that) {
    if (that->type == FrameTetrahedron) {
        printf("T");
    } else if (that->type == FrameCuboid) {
        printf("C");
    }
    printf("o(");
    for (int i = 0;
        i < 2;
        ++i) {

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

```

```

        if (i < 1)
            printf(",");

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

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

        }
    }
    printf(")");
}

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

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

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

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

        }
    }
    printf(")");
}

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

```

```

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

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

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

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

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

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

    }
}
printf(")");
}

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

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

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

```

```

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

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

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

        }
    }
    printf(")");
}

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

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

        res *= base;

    }
    return res;
}

```

4.2 FMB

4.2.1 2D static

Header

```

#ifndef __FMB2D_H_
#define __FMB2D_H_

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

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

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

```

```

// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
// The resulting AABB is given in tho's local coordinates system
bool FMBTestIntersection2D(
    const Frame2D* const that,
    const Frame2D* const tho,
    AABB2D* const bdgBox);

#endif

Body

#include "fmb2d.h"

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

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

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

#define FST_VAR 0
#define SND_VAR 1

#define EPSILON 0.0000001

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

// Eliminate the first 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 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 which has been reduced to only one variable
// and store them in the iVar-th axis of the
// AABB bdgBox
// (M arrangement is [iRow][iCol])
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBoundLastVar2D(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    AABB2D* const bdgBox);

// Get the bounds of the iVar-th variable in the nbRows rows
// system M.X<=Y where the iVar-th variable is on the first column
// and store them in the iVar-th axis of the AABB bdgBox

```

```

// (M arrangement is [iRow][iCol])
void GetBoundVar2D(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    const int nbCols,
    AAB2D* const bdgBox);

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

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

    // Initialize the number of rows in the result system
    int nbResRows = 0;

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

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

        // Shortcuts
        const double fabsMIRowIVar = fabs(M[iRow][0]);

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

            // Shortcuts
            const double* MiRow = M[iRow];
            const int sgnMIRowIVar = sgn(MiRow[0]);
            const 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][0]) &&
                    fabs(M[jRow][0]) > EPSILON) {

                    // Shortcuts
                    const double* MjRow = M[jRow];
                    const double fabsMjRow = fabs(MjRow[0]);

```

```

// 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 = 1;
     iCol < nbCols;
     ++iCol ) {

    Mp[nbResRows][iCol - 1] =
        MiRow[iCol] / fabsMIRowIVar +
        MjRow[iCol] / fabsMjRow;

    // Update the sum of the negative coefficient
    sumNegCoeff += neg(Mp[nbResRows][iCol - 1]);

}

// Update the right side of the inequality
Yp[nbResRows] =
    YIRowDivideByFabsMIRowIVar +
    Y[jRow] / fabsMjRow;

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

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

}

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

}

}

}

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

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

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

    // If the coefficient of the eliminated variable is null on
    // this row
    if (fabs(MiRow[0]) < EPSILON) {

```



```

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

// Copy this row into the result system excluding the eliminated
// variable
for (int iCol = 1;
     iCol < nbCols;
     ++iCol) {

    MpnbResRows[iCol - 1] = MiRow[iCol];

}

Yp[nbResRows] = Y[iRow];

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

}

}

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

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

}

// Get the bounds of the iVar-th variable in the nbRows rows
// system  $M.X \leq Y$  which has been reduced to only one variable
// and store them in the iVar-th axis of the
// AABB bdgBox
// (M arrangement is [iRow][iCol])
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBoundLastVar2D(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    AABB2D* const bdgBox) {

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

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

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

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

        // If this row has been reduced to the variable in argument

```

```

// and it has a strictly positive coefficient
if (MjRowiVar > EPSILON) {

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

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

        // Update the maximum bound
        *max = y;

    }

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

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

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

        // Update the minimum bound
        *min = y;

    }

}

}

}

// Get the bounds of the iVar-th variable in the nbRows rows
// system  $M.X \leq Y$  where the iVar-th variable is on the first column
// and store them in the iVar-th axis of the AABBB bdgBox
// (M arrangement is [iRow][iCol])
void GetBoundVar2D(
    const int iVar,
    const double (*M)[2],
    const double* Y,
    const int nbRows,
    const int nbCols,
    AABBB2D* const bdgBox) {

    // Shortcuts
    double* bdgBoxMin = bdgBox->min;
    double* bdgBoxMax = bdgBox->max;

    // Initialize the bounds
    bdgBoxMin[iVar] = 0.0;
    bdgBoxMax[iVar] = 1.0;

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

        // Shortcuts
        const double* MRow = M[iRow];

```

```

double fabsMIRowIVar = fabs(MIRow[0]);

// If the coefficient of the first variable on this row is not null
if (fabsMIRowIVar > EPSILON) {

    // Declare two variables to memorize the min and max of the
    // requested variable in this row
    double min = -1.0 * Y[iRow];
    double max = Y[iRow];

    // Loop on columns except the first one which is the one of the
    // requested variable
    for (int iCol = 1;
        iCol < nbCols;
        ++iCol) {

        if (MIRow[iCol] > EPSILON) {
            min += MIRow[iCol] * bdgBoxMin[iCol + iVar];
            max -= MIRow[iCol] * bdgBoxMin[iCol + iVar];
        } else if (MIRow[iCol] < EPSILON) {
            min += MIRow[iCol] * bdgBoxMax[iCol + iVar];
            max -= MIRow[iCol] * bdgBoxMax[iCol + iVar];
        }

    }

    min /= -1.0 * MIRow[0];
    max /= MIRow[0];
    if (bdgBoxMin[iVar] > min) {

        bdgBoxMin[iVar] = min;

    }
    if (bdgBoxMax[iVar] < max) {

        bdgBoxMax[iVar] = max;

    }

}

}

}

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

    // Get the projection of the Frame 'tho' in Frame 'that' coordinates
    // system

```

```

Frame2D thoProj;
Frame2DImportFrame(that, tho, &thoProj);

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

// Create the inequality system

//  $-\text{sum\_iC\_j}, iX_i \leq 0_j$ 
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) {

    //  $\text{sum\_iC\_j}, iX_i \leq 1.0 - 0_j$ 
    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 {

    //  $\text{sum\_j}(\text{sum\_iC\_j}, iX_i) \leq 1.0 - \text{sum\_iO\_i}$ 
    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) {

    //  $X_i \leq 1.0$ 
    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 the
AABB2D bdgBoxLocal = {
    .min = {0.0, 0.0},
    .max = {0.0, 0.0}
};

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

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

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

    // The two Frames are not in intersection

```

```

        return false;
    }

    // Get the bounds for the remaining second variable
    GetBoundLastVar2D(
        SND_VAR,
        Mp,
        Yp,
        nbRowsP,
        &bdgBoxLocal);

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

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

    // Else, if the bounds are consistent here it means
    // the two Frames are in intersection.
    // If the user has requested for the resulting bounding box
    } else if (bdgBox != NULL) {

        // Get the bounds of the first variable from the bounds of the
        // second one

        GetBoundVar2D(
            FST_VAR,
            M,
            Y,
            nbRows,
            2,
            &bdgBoxLocal);

        // Memorize the result
        *bdgBox = bdgBoxLocal;

    }

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

```

4.2.2 3D static

Header

```

#ifndef __FMB3D_H_
#define __FMB3D_H_

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

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

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

```

```

// If bdgBox is null, the result AABB is not memorized (to use if
// unnecessary and want to speed up the algorithm)
// The resulting AABB may be larger than the smallest possible AABB
// The resulting AABB of FMBTestIntersection(A,B) may be different
// of the resulting AABB of FMBTestIntersection(B,A)
// The resulting AABB is given in tho's local coordinates system
bool FMBTestIntersection3D(
    const Frame3D* const that,
    const Frame3D* const tho,
    AABB3D* const bdgBox);

#endif

Body

#include "fmb3d.h"

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

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

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

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

#define EPSILON 0.0000001

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

// Eliminate the first 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 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 which has been reduced to only one variable
// and store them in the iVar-th axis of the
// AABB bdgBox
// (M arrangement is [iRow][iCol])
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBoundLastVar3D(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    AABB3D* const bdgBox);

```

```

// Get the bounds of the iVar-th variable in the nbRows rows
// system  $M.X \leq Y$  where the iVar-th variable is on the first column
// and store them in the iVar-th axis of the AABBB bdgBox
// (M arrangement is [iRow][iCol])
void GetBoundVar3D(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    AABBB3D* const bdgBox);

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

// Eliminate the first variable in the system  $M.X \leq Y$ 
// 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 double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[3],
    double* Yp,
    int* const nbRemainRows) {

    // Initialize the number of rows in the result system
    int nbResRows = 0;

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

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

        // Shortcuts
        const double fabsMIRowIVar = fabs(M[iRow][0]);

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

            // Shortcuts
            const double* MiRow = M[iRow];
            const int sgnMIRowIVar = sgn(MiRow[0]);
            const 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][0]) &&
                    fabs(M[jRow][0]) > EPSILON) {

```



```

// Shortcuts
const double* MjRow = M[jRow];
const double fabsMjRow = fabs(MjRow[0]);

// 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 = 1;
     iCol < nbCols;
     ++iCol ) {

    Mp[nbResRows][iCol - 1] =
        MiRow[iCol] / fabsMIRowIVar +
        MjRow[iCol] / fabsMjRow;

    // Update the sum of the negative coefficient
    sumNegCoeff += neg(Mp[nbResRows][iCol - 1]);

}

// Update the right side of the inequality
Yp[nbResRows] =
    YIRowDivideByFabsMIRowIVar +
    Y[jRow] / fabsMjRow;

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

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

}

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

}

}

}

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

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

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

```

```

// If the coefficient of the eliminated variable is null on
// this row
if (fabs(MiRow[0]) < EPSILON) {

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

    // Copy this row into the result system excluding the eliminated
    // variable
    for (int iCol = 1;
        iCol < nbCols;
        ++iCol) {

        MpnbResRows[iCol - 1] = MiRow[iCol];

    }

    Yp[nbResRows] = Y[iRow];

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

}

}

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

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

}

// Get the bounds of the iVar-th variable in the nbRows rows
// system M.X<=Y which has been reduced to only one variable
// and store them in the iVar-th axis of the
// AABBB bdgBox
// (M arrangement is [iRow][iCol])
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBoundLastVar3D(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    AABBB3D* const bdgBox) {

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

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

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

        // Shortcut

```

```

double MjRowiVar = M[jRow][0];

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

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

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

        // Update the maximum bound
        *max = y;

    }

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

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

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

        // Update the minimum bound
        *min = y;

    }

}

}

}

// Get the bounds of the iVar-th variable in the nbRows rows
// system  $M.X \leq Y$  where the iVar-th variable is on the first column
// and store them in the iVar-th axis of the AABB bdgBox
// (M arrangement is [iRow][iCol])
void GetBoundVar3D(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    AABB3D* const bdgBox) {

    // Shortcuts
    double* bdgBoxMin = bdgBox->min;
    double* bdgBoxMax = bdgBox->max;

    // Initialize the bounds
    bdgBoxMin[iVar] = 0.0;
    bdgBoxMax[iVar] = 1.0;

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

```

```

// Shortcuts
const double* MIRow = M[iRow];
double fabsMIRowIVar = fabs(MIRow[0]);

// If the coefficient of the first variable on this row is not null
if (fabsMIRowIVar > EPSILON) {

    // Declare two variables to memorize the min and max of the
    // requested variable in this row
    double min = -1.0 * Y[iRow];
    double max = Y[iRow];

    // Loop on columns except the first one which is the one of the
    // requested variable
    for (int iCol = 1;
        iCol < nbCols;
        ++iCol) {

        if (MIRow[iCol] > EPSILON) {
            min += MIRow[iCol] * bdgBoxMin[iCol + iVar];
            max -= MIRow[iCol] * bdgBoxMin[iCol + iVar];
        } else if (MIRow[iCol] < EPSILON) {
            min += MIRow[iCol] * bdgBoxMax[iCol + iVar];
            max -= MIRow[iCol] * bdgBoxMax[iCol + iVar];
        }

    }

    min /= -1.0 * MIRow[0];
    max /= MIRow[0];
    if (bdgBoxMin[iVar] > min) {

        bdgBoxMin[iVar] = min;

    }
    if (bdgBoxMax[iVar] < max) {

        bdgBoxMax[iVar] = max;

    }

}

}

}

// 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<=0_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<=1.0-0_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_i0_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 the
AABB3D bdgBoxLocal = {
    .min = {0.0, 0.0, 0.0},
    .max = {0.0, 0.0, 0.0}
};

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

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

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

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

}

// Declare variables to eliminate the second variable
// The size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mpp[624][3];
//double Ypp[624];
double Mpp[55][3];
double Ypp[55];
int nbRowsPP;

```

```

// Eliminate the second variable (which is the first in the new system)
inconsistency =
    ElimVar3D(
        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
GetBoundLastVar3D(
    THD_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

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

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

}

// Else, if the bounds are consistent here it means
// the two Frames are in intersection.
// If the user has requested for the resulting bounding box
} else if (bdgBox != NULL) {

    // Get the bounds of the other variables

    GetBoundVar3D(
        SND_VAR,
        Mp,
        Yp,
        nbRowsP,
        2,
        &bdgBoxLocal);

    GetBoundVar3D(
        FST_VAR,
        M,
        Y,
        nbRows,
        3,
        &bdgBoxLocal);

    // Memorize the result
    *bdgBox = bdgBoxLocal;

}

```



```

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

```

4.2.3 2D dynamic

Header

```

#ifndef __FMB2DT_H_
#define __FMB2DT_H_

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

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

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

#endif

```

Body

```

#include "fmb2dt.h"

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

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

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

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

#define EPSILON 0.0000001

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

// Eliminate the first 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 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  $M.X \leq Y$  which has been reduced to only one variable
// and store them in the iVar-th axis of the
// AABBBdgBox
// (M arrangement is [iRow][iCol])
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBoundLastVar2DTime(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    AABBBdgBox* const bdgBox);

// Get the bounds of the iVar-th variable in the nbRows rows
// system  $M.X \leq Y$  where the iVar-th variable is on the first column
// and store them in the iVar-th axis of the AABBBdgBox
// (M arrangement is [iRow][iCol])
void GetBoundVar2DTime(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    AABBBdgBox* const bdgBox);

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

// Eliminate the first variable in the system  $M.X \leq Y$ 
// 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 double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[3],
    double* Yp,
    int* const nbRemainRows) {

    // Initialize the number of rows in the result system
    int nbResRows = 0;

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

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

```

```

// Shortcuts
const double fabsMIRowIVar = fabs(M[iRow][0]);

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

    // Shortcuts
    const double* MiRow = M[iRow];
    const int sgnMIRowIVar = sgn(MiRow[0]);
    const 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][0]) &&
            fabs(M[jRow][0]) > EPSILON) {

            // Shortcuts
            const double* MjRow = M[jRow];
            const double fabsMjRow = fabs(MjRow[0]);

            // 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 = 1;
                iCol < nbCols;
                ++iCol ) {

                Mp[nbResRows][iCol - 1] =
                    MiRow[iCol] / fabsMIRowIVar +
                    MjRow[iCol] / fabsMjRow;

                // Update the sum of the negative coefficient
                sumNegCoeff += neg(Mp[nbResRows][iCol - 1]);

            }

            // Update the right side of the inequality
            Yp[nbResRows] =
                YIRowDivideByFabsMIRowIVar +
                Y[jRow] / fabsMjRow;

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

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

            }

        }

    }

}

```

```

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

    }

}

}

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

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

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

    // If the coefficient of the eliminated variable is null on
    // this row
    if (fabs(MiRow[0]) < EPSILON) {

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

        // Copy this row into the result system excluding the eliminated
        // variable
        for (int iCol = 1;
             iCol < nbCols;
             ++iCol) {

            MpnbResRows[iCol - 1] = MiRow[iCol];

        }

        Yp[nbResRows] = Y[iRow];

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

    }

}

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

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

}

// Get the bounds of the iVar-th variable in the nbRows rows
// system  $M.X \leq Y$  which has been reduced to only one variable
// and store them in the iVar-th axis of the
// AABB bdgBox
// (M arrangement is [iRow][iCol])

```

```

// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBoundLastVar2DTime(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    AABB2DTime* const bdgBox) {

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

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

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

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

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

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

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

                // Update the maximum bound
                *max = y;

            }

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

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

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

                    // Update the minimum bound
                    *min = y;

                }

            }

        }

    }

    // Get the bounds of the iVar-th variable in the nbRows rows

```

```

// system  $M.X \leq Y$  where the iVar-th variable is on the first column
// and store them in the iVar-th axis of the AABBB bdgBox
// (M arrangement is [iRow][iCol])
void GetBoundVar2DTime(
    const int iVar,
    const double (*M)[3],
    const double* Y,
    const int nbRows,
    const int nbCols,
    AABBB2DTime* const bdgBox) {

    // Shortcuts
    double* bdgBoxMin = bdgBox->min;
    double* bdgBoxMax = bdgBox->max;

    // Initialize the bounds
    bdgBoxMin[iVar] = 0.0;
    bdgBoxMax[iVar] = 1.0;

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

        // Shortcuts
        const double* MRow = M[iRow];
        double fabsMRowIVar = fabs(MRow[0]);

        // If the coefficient of the first variable on this row is not null
        if (fabsMRowIVar > EPSILON) {

            // Declare two variables to memorize the min and max of the
            // requested variable in this row
            double min = -1.0 * Y[iRow];
            double max = Y[iRow];

            // Loop on columns except the first one which is the one of the
            // requested variable
            for (int iCol = 1;
                iCol < nbCols;
                ++iCol) {

                if (MRow[iCol] > EPSILON) {
                    min += MRow[iCol] * bdgBoxMin[iCol + iVar];
                    max -= MRow[iCol] * bdgBoxMin[iCol + iVar];
                } else if (MRow[iCol] < EPSILON) {
                    min += MRow[iCol] * bdgBoxMax[iCol + iVar];
                    max -= MRow[iCol] * bdgBoxMax[iCol + iVar];
                }
            }

            min /= -1.0 * MRow[0];
            max /= MRow[0];
            if (bdgBoxMin[iVar] > min) {

                bdgBoxMin[iVar] = min;
            }
            if (bdgBoxMax[iVar] < max) {

                bdgBoxMax[iVar] = max;
            }
        }
    }
}

```

```

    }

}

}

}

// 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<=0_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<=1.0-0_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_i0_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 the
AABB2DTime bdgBoxLocal = {
    .min = {0.0, 0.0, 0.0},
    .max = {0.0, 0.0, 0.0}
};

// Declare variables to eliminate the first variable
// The size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mp[35][3];
//double Yp[35];
double Mp[13][3];
double Yp[13];
int nbRowsP;

// Eliminate the first variable in the original system
bool inconsistency =
    ElimVar2DTime(
        M,
        Y,
        nbRows,
        3,
        Mp,
        Yp,
        &nbRowsP);

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

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

}

// Declare variables to eliminate the second variable
// The size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mpp[342][3];

```

```

//double Ypp[342];
double Mpp[21][3];
double Ypp[21];
int nbRowsPP;

// Eliminate the second variable (which is the first in the new system)
inconsistency =
    ElimVar2DTime(
        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
GetBoundLastVar2DTime(
    THD_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

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

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

}

// Else, if the bounds are consistent here it means
// the two Frames are in intersection.
// If the user has requested for the resulting bounding box
} else if (bdgBox != NULL) {

    // Get the bounds of the other variables

    GetBoundVar2DTime(
        SND_VAR,
        Mp,
        Yp,
        nbRowsP,
        2,
        &bdgBoxLocal);

    GetBoundVar2DTime(
        FST_VAR,
        M,
        Y,
        nbRows,
        3,
        &bdgBoxLocal);

    // Memorize the result

```

```

        *bdgBox = bdgBoxLocal;

    }

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

}

```

4.2.4 3D dynamic

Header

```

#ifndef __FMB3DT_H_
#define __FMB3DT_H_

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

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

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

#endif

```

Body

```

#include "fmb3dt.h"

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

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

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

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

#define EPSILON 0.0000001

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

// Eliminate the first 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 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 which has been reduced to only one variable
// and store them in the iVar-th axis of the
// AABB bdgBox
// (M arrangement is [iRow][iCol])
// May return inconsistent values (max < min), which would
// mean the system has no solution
void GetBoundLastVar3DTime(
    const int iVar,
    const double (*M)[4],
    const double* Y,
    const int nbRows,
    AABB3DTime* const bdgBox);

// Get the bounds of the iVar-th variable in the nbRows rows
// system M.X<=Y where the iVar-th variable is on the first column
// and store them in the iVar-th axis of the AABB bdgBox
// (M arrangement is [iRow][iCol])
void GetBoundVar3DTime(
    const int iVar,
    const double (*M)[4],
    const double* Y,
    const int nbRows,
    const int nbCols,
    AABB3DTime* const bdgBox);

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

// Eliminate the first 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 double (*M)[4],
    const double* Y,
    const int nbRows,
    const int nbCols,
    double (*Mp)[4],
    double* Yp,
    int* const nbRemainRows) {

    // Initialize the number of rows in the result system
    int nbResRows = 0;

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

```

```

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

    // Shortcuts
    const double fabsMIRowIVar = fabs(M[iRow][0]);

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

        // Shortcuts
        const double* MiRow = M[iRow];
        const int sgnMIRowIVar = sgn(MiRow[0]);
        const 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][0]) &&
                fabs(M[jRow][0]) > EPSILON) {

                // Shortcuts
                const double* MjRow = M[jRow];
                const double fabsMjRow = fabs(MjRow[0]);

                // 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 = 1;
                    iCol < nbCols;
                    ++iCol ) {

                    Mp[nbResRows][iCol - 1] =
                        MiRow[iCol] / fabsMIRowIVar +
                        MjRow[iCol] / fabsMjRow;

                    // Update the sum of the negative coefficient
                    sumNegCoeff += neg(Mp[nbResRows][iCol - 1]);

                }

                // Update the right side of the inequality
                Yp[nbResRows] =
                    YIRowDivideByFabsMIRowIVar +
                    Y[jRow] / fabsMjRow;

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

```

```

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

    }

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

}

}

}

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

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

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

    // If the coefficient of the eliminated variable is null on
    // this row
    if (fabs(MiRow[0]) < EPSILON) {

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

        // Copy this row into the result system excluding the eliminated
        // variable
        for (int iCol = 1;
             iCol < nbCols;
             ++iCol) {

            MpnbResRows[iCol - 1] = MiRow[iCol];

        }

        Yp[nbResRows] = Y[iRow];

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

    }

}

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

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

}

```

```

// Get the bounds of the iVar-th variable in the nbRows rows
// system  $M.X \leq Y$  and store them in the iVar-th axis of the
// AABBB 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 GetBoundLastVar3DTime(
    const int iVar,
    const double (*M)[4],
    const double* Y,
    const int nbRows,
    AABBB3DTime* const bdgBox) {

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

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

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

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

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

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

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

                // Update the maximum bound
                *max = y;

            }

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

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

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

                // Update the minimum bound
                *min = y;

            }

        }

    }

}

```

```

    }

}

// Get the bounds of the iVar-th variable in the nbRows rows
// system  $M.X \leq Y$  where the iVar-th variable is on the first column
// and store them in the iVar-th axis of the AABB bdgBox
// (M arrangement is [iRow][iCol])
void GetBoundVar3DTime(
    const int iVar,
    const double (*M)[4],
    const double* Y,
    const int nbRows,
    const int nbCols,
    AABB3DTime* const bdgBox) {

    // Shortcuts
    double* bdgBoxMin = bdgBox->min;
    double* bdgBoxMax = bdgBox->max;

    // Initialize the bounds
    bdgBoxMin[iVar] = 0.0;
    bdgBoxMax[iVar] = 1.0;

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

        // Shortcuts
        const double* MIRow = M[iRow];
        double fabsMIRowIVar = fabs(MIRow[0]);

        // If the coefficient of the first variable on this row is not null
        if (fabsMIRowIVar > EPSILON) {

            // Declare two variables to memorize the min and max of the
            // requested variable in this row
            double min = -1.0 * Y[iRow];
            double max = Y[iRow];

            // Loop on columns except the first one which is the one of the
            // requested variable
            for (int iCol = 1;
                iCol < nbCols;
                ++iCol) {

                if (MIRow[iCol] > EPSILON) {
                    min += MIRow[iCol] * bdgBoxMin[iCol + iVar];
                    max -= MIRow[iCol] * bdgBoxMin[iCol + iVar];
                } else if (MIRow[iCol] < EPSILON) {
                    min += MIRow[iCol] * bdgBoxMax[iCol + iVar];
                    max -= MIRow[iCol] * bdgBoxMax[iCol + iVar];
                }

            }

        }

        min /= -1.0 * MIRow[0];
        max /= MIRow[0];
        if (bdgBoxMin[iVar] > min) {

```



```

        bdgBoxMin[iVar] = min;

    }
    if (bdgBoxMax[iVar] < max) {

        bdgBoxMax[iVar] = max;

    }

}

}

}

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

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

    M[2][0] = -thoProj.comp[0][2];
    M[2][1] = -thoProj.comp[1][2];
    M[2][2] = -thoProj.comp[2][2];

```

```

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

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

if (that->type == FrameCuboid) {

    // V_jT+sum_iC_j, iX_i<=1.0-0_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_i0_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 = {
    .min = {0.0, 0.0, 0.0, 0.0},
    .max = {0.0, 0.0, 0.0, 0.0}
};

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

// Eliminate the first variable in the original system
bool inconsistency =
    ElimVar3DTime(
        M,
        Y,
        nbRows,
        4,
        Mp,
        Yp,
        &nbRowsP);

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

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

}

// Declare variables to eliminate the second variable
// The size of the array given in the doc is a majoring value.
// Instead I use a smaller value which has proven to be sufficient
// during tests, validation and qualification, to avoid running
// into the heap limit and to optimize slightly the performance
//double Mpp[1056][4];
//double Ypp[1056];
double Mpp[57][4];
double Ypp[57];
int nbRowsPP;

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

```

```

        Yp,
        nbRowsP,
        3,
        Mpp,
        Ypp,
        &nbRowsPP);

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

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

}

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

// Eliminate the third variable (which is the first in the new system)
inconsistency =
    ElimVar3DTime(
        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
GetBoundLastVar3DTime(
    FOR_VAR,
    Mppp,
    Yppp,
    nbRowsPPP,
    &bdgBoxLocal);

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

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

}

// Else, if the bounds are consistent here it means
// the two Frames are in intersection.
// If the user has requested for the resulting bounding box

```

```

} else if (bdgBox != NULL) {

    // Get the bounds of the other variables

    GetBoundVar3DTime(
        THD_VAR,
        Mpp,
        Ypp,
        nbRowsPP,
        2,
        &bdgBoxLocal);

    GetBoundVar3DTime(
        SND_VAR,
        Mp,
        Yp,
        nbRowsP,
        3,
        &bdgBoxLocal);

    GetBoundVar3DTime(
        FST_VAR,
        M,
        Y,
        nbRows,
        4,
        &bdgBoxLocal);

    // Memorize the result
    *bdgBox = bdgBoxLocal;

}

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

}

```

5 Minimal example of use

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

5.1 2D static

```

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

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

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

```

```

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

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

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

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

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

    printf("Intersection detected in AABB ");

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

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

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

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

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

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

        }

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

```

```

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

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

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

    }

}

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

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

    printf("No intersection.\n");

}

return 0;

}

```

5.2 3D static

```

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

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

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

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

    double origQ3D[3] = {0.0, 0.0, 0.0};
    double compQ3D[3][3] = {
        {1.0, 0.0, 0.0},
        {0.0, 1.0, 0.0},
        {0.0, 0.0, 1.0}};
    Frame3D Q3D =
        Frame3DCreateStatic(
            FrameTetrahedron,
            origQ3D,
            compQ3D);
}

```



```

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

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

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

    printf("Intersection detected in AABB ");

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

    // Clip with the AABB of 'Q3D' and 'P3D' to improve results
    for (int iAxis = 3;
        iAxis--;) {

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

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

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

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

        }

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

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

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

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

        }

    }

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

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

    printf("No intersection.\n");

}

```

```

    return 0;
}

```

5.3 2D dynamic

```

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

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

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

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

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

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

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

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

        printf("Intersection detected in AABB ");

        // Export the local bounding box toward the real coordinates
        // system
        AABB2DTime bdgBox2DTime;
        Frame2DTimeExportBdgBox(

```

```

        &Q2DTime,
        &bdgBox2DTimeLocal,
        &bdgBox2DTime);

// Clip with the AABB of 'Q2DTime' and 'P2DTime' to improve results
for (int iAxis = 3;
     iAxis--;) {

    if (bdgBox2DTime.min[iAxis] < P2DTime.bdgBox.min[iAxis]) {

        bdgBox2DTime.min[iAxis] = P2DTime.bdgBox.min[iAxis];

    }
    if (bdgBox2DTime.max[iAxis] > P2DTime.bdgBox.max[iAxis]) {

        bdgBox2DTime.max[iAxis] = P2DTime.bdgBox.max[iAxis];

    }

    if (bdgBox2DTime.min[iAxis] < Q2DTime.bdgBox.min[iAxis]) {

        bdgBox2DTime.min[iAxis] = Q2DTime.bdgBox.min[iAxis];

    }
    if (bdgBox2DTime.max[iAxis] > Q2DTime.bdgBox.max[iAxis]) {

        bdgBox2DTime.max[iAxis] = Q2DTime.bdgBox.max[iAxis];

    }

}

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

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

    printf("No intersection.\n");

}

return 0;
}

```

5.4 3D dynamic

```

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

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

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

    // Create the two objects to be tested for intersection
    double origP3DTime[3] = {0.0, 0.0, 0.0};
    double speedP3DTime[3] = {0.0, 0.0, 0.0};
}

```

```

double compP3DTime[3][3] = {
    {1.0, 0.0, 0.0}, // First component
    {0.0, 1.0, 0.0}, // Second component
    {0.0, 0.0, 1.0}}; // Third component
Frame3DTime P3DTime =
    Frame3DTimeCreateStatic(
        FrameCuboid,
        origP3DTime,
        speedP3DTime,
        compP3DTime);

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

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

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

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

    printf("Intersection detected in AABB ");

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

    // Clip with the AABB of 'Q3DTime' and 'P3DTime' to improve results
    for (int iAxis = 3;
        iAxis--;) {

        if (bdgBox3DTime.min[iAxis] < P3DTime.bdgBox.min[iAxis]) {

            bdgBox3DTime.min[iAxis] = P3DTime.bdgBox.min[iAxis];

        }
        if (bdgBox3DTime.max[iAxis] > P3DTime.bdgBox.max[iAxis]) {

            bdgBox3DTime.max[iAxis] = P3DTime.bdgBox.max[iAxis];

        }
    }
}

```

```

        if (bdgBox3DTime.min[iAxis] < Q3DTime.bdgBox.min[iAxis]) {
            bdgBox3DTime.min[iAxis] = Q3DTime.bdgBox.min[iAxis];
        }
        if (bdgBox3DTime.max[iAxis] > Q3DTime.bdgBox.max[iAxis]) {
            bdgBox3DTime.max[iAxis] = Q3DTime.bdgBox.max[iAxis];
        }
    }

    AAB3DTimePrint(&bdgBox3DTime);
    printf("\n");

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

    printf("No intersection.\n");

}

return 0;
}

```

6 Unit tests

In this section I introduce the code I've used to test the algorithm and its implementation. The test consists of running the algorithm on a set of cases for which the solution has been computed by hand. The code of the implementation of the SAT algorithm is given in annex (p.243)

6.1 Code

6.1.1 2D static

```

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

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

// Epsilon for numerical precision
#define EPSILON 0.0001

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

```

```

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

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

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

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

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

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

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

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

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

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

```

```

    exit(0);

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

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

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

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

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

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

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

                flag = false;
            }
        }

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

            // Display information
            printf("Succeed\n");
            AABB2DPrint(&bdgBox);
            printf("\n");

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

            // Display information
            printf("Failed\n");
            printf("Expected : ");
            AABB2DPrint(correctBdgBox);
            printf("\n");
        }
    }
}

```

```

        printf("        Got : ");
        AABB2DPrint(&bdgBox);

        // Terminate the unit tests
        exit(0);

    }

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

    // Display information
    printf(" Succeed (no inter)\n");

}

}

printf("\n");

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

}

}

void Test2D(void) {

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

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

    // Execute the unit test on various cases

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

```



```

        true,
        &correctBdgBox);

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

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

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0},
 .comp =
    {{1.0, 0.0},
     {0.0, 1.0}}
};
paramQ = (Param2D)
{.type = FrameCuboid,
 .orig = {0.25, -0.25},
 .comp =
    {{0.5, 0.0},

```

```

        {0.0, 2.0}}
    };
    correctBdgBox = (AABB2D)
    {
        .min = {0.25, 0.0},
        .max = {0.75, 1.0}
    };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

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

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

```

```

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

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

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

```

```

        {{1.0, 0.0},
         {0.0, -1.0}}
    };
    correctBdgBox = (AABB2D)
        {.min = {0.0, 0.0},
         .max = {1.0, 1.0}
        };
    UnitTest2D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

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

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

```

```

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0},
 .comp =
   {{1.0, 0.5},
    {0.5, 1.0}}
};
paramQ = (Param2D)
{.type = FrameCuboid,
 .orig = {1.0, 2.0},
 .comp =
   {{-0.5, -0.5},
    {0.0, -1.0}}
};
correctBdgBox = (AABB2D)
{.min = {0.5, 0.75},
 .max = {1.0, 1.25}
};
UnitTest2D(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

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

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0},
 .comp =
   {{1.0, 0.5},
    {0.5, 1.0}}
};
paramQ = (Param2D)
{.type = FrameTetrahedron,
 .orig = {1.0, 2.0},

```

```

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

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

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

```

```

    &correctBdgBox);

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

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

// -----
paramP = (Param2D)
{.type = FrameCuboid,
 .orig = {0.0, 0.0},
 .comp =
    {{1.0, 0.0},
     {0.0, 1.0}}
};
paramQ = (Param2D)
{.type = FrameTetrahedron,
 .orig = {1.5, 1.5},
 .comp =
    {{-1.5, 0.0},
     {0.0, -1.5}}
};

```

```

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

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

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

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

```



```

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

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

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

    Test2D();

    return 0;
}

```

6.1.2 3D static

```

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

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

// Epsilon for numerical precision
#define EPSILON 0.0001

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

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

    // Create the two Frames

```

```

Frame3D P =
    Frame3DCreateStatic(
        paramP.type,
        paramP.orig,
        paramP.comp);

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

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

        flag = false;
    }
}

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

    // Display information
    printf("Succeed\n");
    AABBB3DPrint(&bdgBox);
    printf("\n");

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

    // Display information
    printf("Failed\n");
    printf("Expected : ");
    AABBB3DPrint(correctBdgBox);
    printf("\n");
    printf("      Got : ");
    AABBB3DPrint(&bdgBox);
    printf("\n");

    // Terminate the unit tests
    exit(0);
}

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

    // Display information

```

```

        printf(" Succeed (no inter)\n");

    }

}

printf("\n");

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

}

}

void Test3D(void) {

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

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

    // Execute the unit test on various cases

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

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

```

```

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

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

// -----
    paramP = (Param3D)
    {
        .type = FrameCuboid,
        .orig = {0.0, 0.0, 0.0},
        .comp =
        {
            {1.0, 0.0, 0.0},
            {0.0, 1.0, 0.0},
            {0.0, 0.0, 1.0}
        }
    };
    paramQ = (Param3D)
    {
        .type = FrameCuboid,
        .orig = {1.5, 1.5, 1.5},
        .comp =
        {
            {-1.0, 0.0, 0.0},
            {0.0, -1.0, 0.0},
            {0.0, 0.0, -1.0}
        }
    };

```

```

    };
    correctBdgBox = (AABB3D)
        {.min = {0.5, 0.5, 0.5},
         .max = {1.0, 1.0, 1.0}
        };
    UnitTest3D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

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

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

    // -----

```

```

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

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

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

```

```

        {{1.0, 0.0, 0.0},
         {0.0, 1.0, 0.0},
         {0.0, 0.0, 1.0}}
    };
    UnitTest3D(
        paramP,
        paramQ,
        false,
        NULL);

    // -----
    paramP = (Param3D)
        {.type = FrameCuboid,
         .orig = {-1.0, -1.0, -1.0},
         .comp =
             {{1.0, 0.0, 0.0},
              {1.0, 1.0, 1.0},
              {0.0, 0.0, 1.0}}
        };
    paramQ = (Param3D)
        {.type = FrameTetrahedron,
         .orig = {0.0, -0.5, 0.0},
         .comp =
             {{1.0, 0.0, 0.0},
              {0.0, 1.0, 0.0},
              {0.0, 0.0, 1.0}}
        };
    correctBdgBox = (AABB3D)
        {.min = {0.0, -0.5, 0.0},
         .max = {0.75, 0.0, 0.75}
        };
    UnitTest3D(
        paramP,
        paramQ,
        true,
        &correctBdgBox);

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

    // -----
    paramP = (Param3D)

```



```

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

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

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

    Test3D();

    return 0;
}

```

6.1.3 2D dynamic

```

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

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

// Epsilon for numerical precision
#define EPSILON 0.0001

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

// Unit test function

```

```

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

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

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

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

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

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

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

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

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

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

```

```

    exit(0);

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

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

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

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

                flag = false;
            }
        }

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

            // Display information
            printf("Succeed\n");
            AABB2DTimePrint(&bdgBox);
            printf("\n");

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

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

            // Terminate the unit tests
            exit(0);
        }

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

        // Display information
        printf(" Succeed (no inter)\n");
    }
}

```

```

        printf("\n");

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

void Test2DTime(void) {

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

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

    // Execute the unit test on various cases

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

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

```

```

UnitTest2DTime(
    paramP,
    paramQ,
    false,
    NULL);

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

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

// -----
paramP = (Param2DTime)

```

```

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

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

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

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

    Test2DTime();

```

```

    return 0;
}

```

6.1.4 3D dynamic

```

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

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

// Epsilon for numerical precision
#define EPSILON 0.0001

// Helper structure to pass arguments to the UnitTest function
typedef struct {
    FrameType type;
    double orig[3];
    double comp[3][3];
    double speed[3];
} Param3DTime;

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

                flag = false;
            }

        }

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

            // Display information
            printf("Succeed\n");
            AABB3DTimePrint(&bdgBox);
            printf("\n");

            // Else, the bounding box wasn't the expected one

```



```

    } else {

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

        // Terminate the unit tests
        exit(0);

    }

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

        // Display information
        printf(" Succeed (no inter)\n");

    }

}

printf("\n");

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

}

}

void Test3DTime(void) {

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

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

    // Execute the unit test on various cases

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

    paramQ = (Param3DTime)
    {
        .type = FrameCuboid,
        .orig = {-1.0, 0.0, 0.0},
        .comp =

```

```

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

// -----
    paramP = (Param3DTime)
        {.type = FrameCuboid,
         .orig = {0.0, 0.0, 0.0},
         .comp =
             {{1.0, 0.0, 0.0},
              {0.0, 1.0, 0.0},
              {0.0, 0.0, 1.0}},
         .speed = {0.0, 0.0, 0.0}
        };
    paramQ = (Param3DTime)
        {.type = FrameCuboid,
         .orig = {-1.01, -1.01, 0.0},
         .comp =
             {{1.0, 0.0, 0.0},
              {0.0, 1.0, 0.0},
              {0.0, 0.0, 1.0}},
         .speed = {1.0, 0.0, 0.0}
        };
    UnitTest3DTime(
        paramP,
        paramQ,
        false,
        NULL);

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

```

```

    true,
    &correctBdgBox);

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

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

// -----

```

```

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

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

}

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

    Test3DTime();

    return 0;
}

```

6.2 Results

6.2.1 2D static

```

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

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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

```

```

minXY(0.500000,0.500000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.000000)

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 (no inter)

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 (no inter)

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
minXY(0.250000,0.000000)-maxXY(0.750000,1.000000)

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
minXY(0.250000,0.000000)-maxXY(0.750000,1.000000)

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
minXY(0.000000,0.250000)-maxXY(1.000000,0.750000)

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
minXY(0.000000,0.250000)-maxXY(1.000000,0.750000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(0.500000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(0.500000,1.000000)

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
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(1.500000,0.000000)-maxXY(1.666667,1.000000)

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
minXY(1.500000,0.500000)-maxXY(2.000000,1.000000)

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
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.500000)

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

```

```

minXY(0.500000,0.500000)-maxXY(1.000000,1.500000)

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,1.000000)-maxXY(1.000000,1.500000)

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
minXY(0.500000,1.000000)-maxXY(1.000000,1.500000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,0.500000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,0.500000)

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
minXY(0.000000,0.000000)-maxXY(0.500000,0.500000)

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
minXY(0.000000,0.000000)-maxXY(0.500000,0.500000)

```

```

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 (no inter)

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 (no inter)

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
minXY(0.000000,0.500000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.500000)-maxXY(1.000000,1.000000)

To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
To(1.010000,1.010000) x(-1.000000,0.000000) y(0.000000,-1.000000)
Succeed (no inter)

To(1.010000,1.010000) x(-1.000000,0.000000) y(0.000000,-1.000000)
against
To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed (no inter)

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

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

To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
against
To(1.010000,1.500000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
Succeed (no inter)

To(1.010000,1.500000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
against
To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
Succeed (no inter)

All unit tests 2D have succeed.

```

6.2.2 3D static

```

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

```



```

(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 (no inter)

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 (no inter)

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
minXYZ(0.500000,0.500000,-1.000000)-maxXYZ(1.000000,1.000000,-0.500000)

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
minXYZ(0.500000,0.500000,-1.000000)-maxXYZ(1.000000,1.000000,-0.500000)

Co(-1.010000,-1.010000,-1.010000) x(1.000000,0.000000,0.000000) y
(1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
(0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed (no inter)

Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
(0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(-1.010000,-1.010000,-1.010000) x(1.000000,0.000000,0.000000) y
(1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
Succeed (no inter)

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

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

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

```

```

Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
  Succeed (no inter)

Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
  Succeed (no inter)

Co(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed
minXYZ(0.000000,-0.500000,0.000000)-maxXYZ(1.000000,0.000000,0.750000)

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

To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
  Succeed (no inter)

To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
  Succeed (no inter)

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

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

All unit tests 3D have succeed.

```

6.2.3 2D dynamic

```

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
against
Co(-1.000000,0.000000) s(-1.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed (no inter)

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 (no inter)

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
against
Co(-1.010000,-1.010000) s(1.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed (no inter)

Co(-1.010000,-1.010000) s(1.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed (no inter)

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
minXYT(-1.000000,0.000000,0.000000)-maxXYT(2.000000,1.000000,1.000000)

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
minXYT(-1.000000,0.000000,0.000000)-maxXYT(1.000000,1.000000,1.000000)

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
against
Co(-1.000000,0.250000) s(4.000000,0.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
Succeed
minXYT(-1.500000,0.000000,0.125000)-maxXYT(2.500000,1.000000,0.500000)

Co(-1.000000,0.250000) s(4.000000,0.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed
minXYT(-0.500000,0.000000,0.125000)-maxXYT(1.500000,1.000000,0.500000)

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)

```

```

against
Co(0.250000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
Succeed
minXYT(0.000000,-1.500000,0.125000)-maxXYT(1.000000,2.500000,0.500000)

Co(0.250000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed
minXYT(0.000000,-0.500000,0.125000)-maxXYT(1.000000,1.500000,0.500000)

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
against
Co(0.900000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
Succeed
minXYT(0.000000,-1.500000,0.125000)-maxXYT(1.400000,2.500000,0.500000)

Co(0.900000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed
minXYT(0.000000,-0.500000,0.125000)-maxXYT(1.400000,1.500000,0.500000)

All unit tests 2DTime have succeed.

```

6.2.4 3D dynamic

```

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(-1.000000,0.000000,0.000000) s(-1.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed (no inter)

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 (no inter)

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(-1.010000,-1.010000,0.000000) s(1.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed (no inter)

```

```

Co(-1.010000,-1.010000,0.000000) s(1.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed (no inter)

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
minXYZT(-1.000000,0.000000,0.000000,0.000000)-maxXYZT
(2.000000,1.000000,1.000000,1.000000)

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
minXYZT(-1.000000,0.000000,0.000000,0.000000)-maxXYZT
(1.000000,1.000000,1.000000,1.000000)

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(-1.000000,0.250000,0.000000) s(4.000000,0.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(-1.500000,0.000000,0.000000,0.125000)-maxXYZT
(2.500000,1.000000,1.000000,0.500000)

Co(-1.000000,0.250000,0.000000) s(4.000000,0.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(-0.500000,0.000000,0.000000,0.125000)-maxXYZT
(1.500000,1.000000,1.000000,0.500000)

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.250000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(0.000000,-1.500000,0.000000,0.125000)-maxXYZT

```

```

(1.000000,2.500000,1.000000,0.500000)

Co(0.250000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(0.000000,-0.500000,0.000000,0.125000)-maxXYZT
(1.000000,1.500000,1.000000,0.500000)

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.900000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(0.000000,-1.500000,0.000000,0.125000)-maxXYZT
(1.400000,2.500000,1.000000,0.500000)

Co(0.900000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(0.000000,-0.500000,0.000000,0.125000)-maxXYZT
(1.400000,1.500000,1.000000,0.500000)

All unit tests 3DTime have succeed.

```

7 Validation against SAT

In this section I introduce the code I've used to validate the algorithm and its implementation. The validation consists of running the FMB algorithm on randomly generated pairs of Frame and check that its result is equal to the one of running the SAT algorithm on the same pair of Frames. The code of the implementation of the SAT algorithm is given in annex (p.243)

7.1 Code

7.1.1 2D static

```

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

```

```

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

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

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

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

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

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void ValidationOnePair2D(
    const Param2D paramP,
    const Param2D paramQ) {

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

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

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

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

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

```



```

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

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

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

    // Stop the validation
    exit(0);

}

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

    // Update the number of intersection
    nbInter++;

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

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

}

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

}

}

void Validate2D(void) {

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

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

    // Initialize the number of intersection and no intersection

```

```

nbInter = 0;
nbNoInter = 0;

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

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

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

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

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

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

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

            }

        }

        param = &paramQ;
    }

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

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

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

        // Run the validation on the two Frames
        ValidationOnePair2D(
            paramP,
            paramQ);

    }

}

// If we reached it means the validation was successfull
// Print results
printf("Validation2D has succeed.\n");
printf("Tested %lu intersections ", nbInter);

```

```

    printf("and %lu no intersections\n", nbNoInter);
}

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

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

    return 0;
}

```

7.1.2 3D static

```

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

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

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

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

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

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

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void ValidationOnePair3D(
    const Param3D paramP,
    const Param3D paramQ) {

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

    Frame3D Q =

```

```

    Frame3DCreateStatic(
        paramQ.type,
        paramQ.orig,
        paramQ.comp);

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

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

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

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

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

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

        // Stop the validation
        exit(0);
    }

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

        // Update the number of intersection
        nbInter++;

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

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

```

```

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

    }

}

void Validate3D(void) {

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

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

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

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

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

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

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

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

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

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

                }

            }

            param = &paramQ;

        }

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

```

```

        paramP.comp[0][2] * paramP.comp[2][1]) +
        paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2] -
        paramP.comp[0][2] * paramP.comp[1][1]);

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

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

        // Run the validation on the two Frames
        ValidationOnePair3D(
            paramP,
            paramQ);

    }

}

// If we reached it means the validation was successfull
// Print results
printf("Validation3D has succeed.\n");
printf("Tested %lu intersections ", nbInter);
printf("and %lu no intersections\n", nbNoInter);

}

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

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

    return 0;
}

```

7.1.3 2D dynamic

```

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

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

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

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

```

```

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

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

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void ValidationOnePair2DTime(
    const Param2DTime paramP,
    const Param2DTime paramQ) {

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

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

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

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

        // Test intersection with FMB
        bool isIntersectingFMB =
            FMBTestIntersection2DTime(
                that,
                tho,
                NULL);

        // Test intersection with SAT
        bool isIntersectingSAT =
            SATTestIntersection2DTime(
                that,
                tho);

        // If the results are different
        if (isIntersectingFMB != isIntersectingSAT) {

            // Print the disagreement
            printf("Validation2D has failed\n");
        }
    }
}

```

```

    Frame2DTimePrint(that);
    printf(" against ");
    Frame2DTimePrint(tho);
    printf("\n");
    printf("FMB : ");
    if (isIntersectingFMB == false)
        printf("no ");
    printf("intersection\n");
    printf("SAT : ");
    if (isIntersectingSAT == false)
        printf("no ");
    printf("intersection\n");

    // Stop the validation
    exit(0);
}

// If the Frames are in intersection
if (isIntersectingFMB == true) {

    // Update the number of intersection
    nbInter++;

// If the Frames are not in intersection
} else {

    // Update the number of no intersection
    nbNoInter++;

}

// Flip the pair of Frames
that = &Q;
tho = &P;

}

}

void Validate2DTime(void) {

    // Initialise the random generator
    srand(time(NULL));

    // Declare two variables to memorize the arguments to the
    // Validation function
    Param2DTime paramP;
    Param2DTime paramQ;

    // Initialize the number of intersection and no intersection
    nbInter = 0;
    nbNoInter = 0;

    // Loop on the tests
    for (unsigned long iTest = NB_TESTS;
        iTest--;) {

        // Create two random Frame definitions
        Param2DTime* param = &paramP;
        for (int iParam = 2;
            iParam--;) {

```



```

        // 50% chance of being a Cuboid or a Tetrahedron
        if (rnd() < 0.5)
            param->type = FrameCuboid;
        else
            param->type = FrameTetrahedron;

        for (int iAxis = 2;
            iAxis--;) {

            param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
            param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            for (int iComp = 2;
                iComp--;) {

                param->comp[iComp][iAxis] =
                    -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            }

        }

        param = &paramQ;
    }

    // Calculate the determinant of the Frames' components matrix
    double detP =
        paramP.comp[0][0] * paramP.comp[1][1] -
        paramP.comp[1][0] * paramP.comp[0][1];

    double detQ =
        paramQ.comp[0][0] * paramQ.comp[1][1] -
        paramQ.comp[1][0] * paramQ.comp[0][1];

    // If the determinants are not null, ie the Frame are not degenerate
    if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

        // Run the validation on the two Frames
        ValidationOnePair2DTime(
            paramP,
            paramQ);

    }

}

// If we reached it means the validation was successfull
// Print results
printf("Validation2DTime has succeed.\n");
printf("Tested %lu intersections ", nbInter);
printf("and %lu no intersections\n", nbNoInter);

}

int main(int argc, char** argv) {

    printf("==== 2D dynamic =====\n");
    Validate2DTime();

    return 0;
}

```

```
}
```

7.1.4 3D dynamic

```
// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>

// Include the FMB and SAT algorithm library
#include "fmb3dt.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of tests of the validation
#define NB_TESTS 1000000

// Helper macro to generate random number in [0.0, 1.0]
#define rnd() (double)(rand())/(double)(RAND_MAX)

// Global variables to count nb of tests resulting in intersection
// and no intersection
unsigned long int nbInter;
unsigned long int nbNoInter;

// Helper structure to pass arguments to the Validation function
typedef struct {
    FrameType type;
    double orig[3];
    double comp[3][3];
    double speed[3];
} Param3DTime;

// Validation function
// Takes two Frame definition as input, run the intersection test on
// them with FMB and SAT, and check the results are identical
void ValidationOnePair3DTime(
    const Param3DTime paramP,
    const Param3DTime paramQ) {

    // Create the two Frames
    Frame3DTime P =
        Frame3DTimeCreateStatic(
            paramP.type,
            paramP.orig,
            paramP.speed,
            paramP.comp);

    Frame3DTime Q =
        Frame3DTimeCreateStatic(
            paramQ.type,
            paramQ.orig,
            paramQ.speed,
            paramQ.comp);

    // Helper variables to loop on the pair (that, tho) and (tho, that)
    Frame3DTime* that = &P;
```

```

Frame3DTime* tho = &Q;

// Loop on pairs of Frames
for (int iPair = 2;
     iPair--;) {

    // Test intersection with FMB
    bool isIntersectingFMB =
        FMBTestIntersection3DTime(
            that,
            tho,
            NULL);

    // Test intersection with SAT
    bool isIntersectingSAT =
        SATTestIntersection3DTime(
            that,
            tho);

    // If the results are different
    if (isIntersectingFMB != isIntersectingSAT) {

        // Print the disagreement
        printf("Validation3D has failed\n");
        Frame3DTimePrint(that);
        printf(" against ");
        Frame3DTimePrint(tho);
        printf("\n");
        printf("FMB : ");
        if (isIntersectingFMB == false)
            printf("no ");
        printf("intersection\n");
        printf("SAT : ");
        if (isIntersectingSAT == false)
            printf("no ");
        printf("intersection\n");

        // Stop the validation
        exit(0);

    }

    // If the Frames are in intersection
    if (isIntersectingFMB == true) {

        // Update the number of intersection
        nbInter++;

    }

    // If the Frames are not in intersection
    } else {

        // Update the number of no intersection
        nbNoInter++;

    }

    // Flip the pair of Frames
    that = &Q;
    tho = &P;

}

```

```

}

void Validate3DTime(void) {

    // Initialise the random generator
    srand(time(NULL));

    // Declare two variables to memorize the arguments to the
    // Validation function
    Param3DTime paramP;
    Param3DTime paramQ;

    // Initialize the number of intersection and no intersection
    nbInter = 0;
    nbNoInter = 0;

    // Loop on the tests
    for (unsigned long iTest = NB_TESTS;
        iTest--;) {

        // Create two random Frame definitions
        Param3DTime* param = &paramP;
        for (int iParam = 2;
            iParam--;) {

            // 50% chance of being a Cuboid or a Tetrahedron
            if (rnd() < 0.5)
                param->type = FrameCuboid;
            else
                param->type = FrameTetrahedron;

            for (int iAxis = 3;
                iAxis--;) {

                param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
                param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

                for (int iComp = 3;
                    iComp--;) {

                    param->comp[iComp][iAxis] =
                        -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

                }

            }

            param = &paramQ;

        }

        // Calculate the determinant of the Frames' components matrix
        double detP =
            paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2] -
            paramP.comp[1][2] * paramP.comp[2][1]) -
            paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2] -
            paramP.comp[0][2] * paramP.comp[2][1]) +
            paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2] -
            paramP.comp[0][2] * paramP.comp[1][1]);

        double detQ =
            paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2] -

```

```

    paramQ.comp[1][2] * paramQ.comp[2][1]) -
    paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2] -
    paramQ.comp[0][2] * paramQ.comp[2][1]) +
    paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2] -
    paramQ.comp[0][2] * paramQ.comp[1][1]);

    // If the determinants are not null, ie the Frame are not degenerate
    if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

        // Run the validation on the two Frames
        ValidationOnePair3DTime(
            paramP,
            paramQ);
    }

}

// If we reached it means the validation was successfull
// Print results
printf("Validation3DTime has succeed.\n");
printf("Tested %lu intersections ", nbInter);
printf("and %lu no intersections\n", nbNoInter);
}

int main(int argc, char** argv) {

    printf("==== 3D dynamic =====\n");
    Validate3DTime();

    return 0;
}

```

7.2 Results

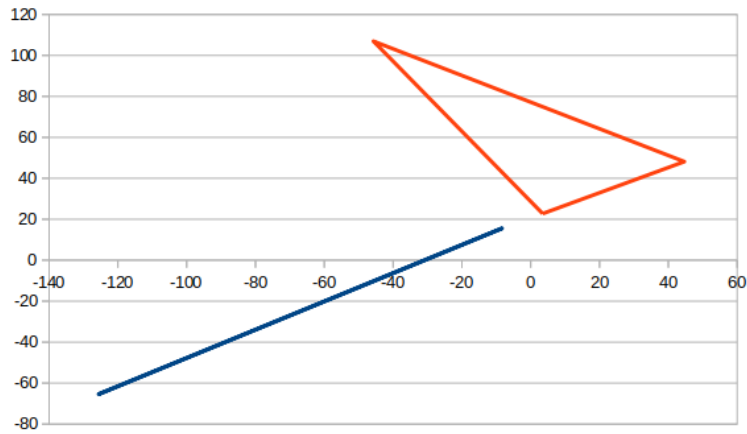
7.2.1 Failures

Validation has failed in one case: when one or both of the frame are degenerated (at least two of there components are colinear). An example is given below for reference:

```

===== 2D static =====
Validation2D has failed
Co(-63.571705,-22.581119) x(55.239119,38.152177) y(-62.031537,-42.843548) against To(3.474294,22.751011)
x(-49.195251,84.166201) y(41.179031,-95.350316)
FMB : intersection
SAT : no intersection

```



This case can be detected and avoided prior to the intersection test by checking the determinant of the frame: degenerated frames have a null determinant. In the example above the determinant of the first frame is equal to -0.001667.

7.2.2 2D static

```
===== 2D static =====
Validation2D has succeed.
Tested 470896 intersections and 1529024 no intersections
```

7.2.3 2D dynamic

```
===== 2D dynamic =====
Validation2DTime has succeed.
Tested 744902 intersections and 1255034 no intersections
```

7.2.4 3D static

```
===== 3D static =====
Validation3D has succeed.
Tested 316364 intersections and 1683634 no intersections
```

7.2.5 3D dynamic

```
===== 3D dynamic =====
Validation3DTime has succeed.
Tested 523984 intersections and 1476016 no intersections
```

8 Qualification against SAT

In this section I introduce the code I've used to qualify the algorithm and its implementation. The qualification consists of running the FMB algorithm

on randomly generated pairs of Frame, and check its execution time against the one of running the SAT algorithm on the same pair of Frames.

8.1 Code

8.1.1 2D static

```
// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>

// Include FMB and SAT algorithm library
#include "fmb2d.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 2000
// 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
    srand(time(NULL));

    // Open the files to save the results
    FILE* fp = fopen("../Results/qualification2D.txt", "w");
    FILE* fpCC = fopen("../Results/qualification2DCC.txt", "w");
    FILE* fpCT = fopen("../Results/qualification2DCT.txt", "w");
    FILE* fpTC = fopen("../Results/qualification2DTC.txt", "w");
    FILE* fpTT = fopen("../Results/qualification2DTT.txt", "w");

    // Loop on runs
    for (int iRun = 0;
        iRun < NB_RUNS;
        ++iRun) {

        // Ratio intersection/no intersection for the displayed results
        double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

        // Initialize counters
        minInter = 0.0;
        maxInter = 0.0;
        sumInter = 0.0;
        countInter = 0;
        minNoInter = 0.0;
        maxNoInter = 0.0;
        sumNoInter = 0.0;
        countNoInter = 0;

        minInterCC = 0.0;
        maxInterCC = 0.0;
        sumInterCC = 0.0;
        countInterCC = 0;
        minNoInterCC = 0.0;
        maxNoInterCC = 0.0;
        sumNoInterCC = 0.0;
        countNoInterCC = 0;

        minInterCT = 0.0;
        maxInterCT = 0.0;
        sumInterCT = 0.0;
        countInterCT = 0;
        minNoInterCT = 0.0;
        maxNoInterCT = 0.0;
        sumNoInterCT = 0.0;
        countNoInterCT = 0;

        minInterTC = 0.0;

```

```

maxInterTC = 0.0;
sumInterTC = 0.0;
countInterTC = 0;
minNoInterTC = 0.0;
maxNoInterTC = 0.0;
sumNoInterTC = 0.0;
countNoInterTC = 0;

minInterTT = 0.0;
maxInterTT = 0.0;
sumInterTT = 0.0;
countInterTT = 0;
minNoInterTT = 0.0;
maxNoInterTT = 0.0;
sumNoInterTT = 0.0;
countNoInterTT = 0;

// Declare two variables to memorize the arguments to the
// Qualification function
Param2D paramP;
Param2D paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

    // Create two random Frame definitions
    Param2D* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

        // 50% chance of being a Cuboid or a Tetrahedron
        if (rnd() < 0.5)
            param->type = FrameCuboid;
        else
            param->type = FrameTetrahedron;

        for (int iAxis = 2;
             iAxis--;) {

            param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            for (int iComp = 2;
                 iComp--;) {

                param->comp[iComp][iAxis] =
                    -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            }

        }

        param = &paramQ;

    }

    // Calculate the determinant of the Frames' components matrix

    double detP =
        paramP.comp[0][0] * paramP.comp[1][1] -
        paramP.comp[1][0] * paramP.comp[0][1];

```

```

double detQ =
    paramQ.comp[0][0] * paramQ.comp[1][1] -
    paramQ.comp[1][0] * paramQ.comp[0][1];

// If the determinants are not null, ie the Frame are not degenerate
if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Qualification2DStatic(
        paramP,
        paramQ);

}

}

// Display the results
if (iRun == 0) {

    fprintf(fp, "percPairInter,");
    fprintf(fp, "countInterTo,countNoInterTo,");
    fprintf(fp, "minInterTo,avgInterTo,maxInterTo,");
    fprintf(fp, "minNoInterTo,avgNoInterTo,maxNoInterTo,");
    fprintf(fp, "minTotalTo,avgTotalTo,maxTotalTo\n");

    fprintf(fpCC, "percPairInter,");
    fprintf(fpCC, "countInterCC,countNoInterCC,");
    fprintf(fpCC, "minInterCC,avgInterCC,maxInterCC,");
    fprintf(fpCC, "minNoInterCC,avgNoInterCC,maxNoInterCC,");
    fprintf(fpCC, "minTotalCC,avgTotalCC,maxTotalCC\n");

    fprintf(fpCT, "percPairInter,");
    fprintf(fpCT, "countInterCT,countNoInterCT,");
    fprintf(fpCT, "minInterCT,avgInterCT,maxInterCT,");
    fprintf(fpCT, "minNoInterCT,avgNoInterCT,maxNoInterCT,");
    fprintf(fpCT, "minTotalCT,avgTotalCT,maxTotalCT\n");

    fprintf(fpTC, "percPairInter,");
    fprintf(fpTC, "countInterTC,countNoInterTC,");
    fprintf(fpTC, "minInterTC,avgInterTC,maxInterTC,");
    fprintf(fpTC, "minNoInterTC,avgNoInterTC,maxNoInterTC,");
    fprintf(fpTC, "minTotalTC,avgTotalTC,maxTotalTC\n");

    fprintf(fpTT, "percPairInter,");
    fprintf(fpTT, "countInterTT,countNoInterTT,");
    fprintf(fpTT, "minInterTT,avgInterTT,maxInterTT,");
    fprintf(fpTT, "minNoInterTT,avgNoInterTT,maxNoInterTT,");
    fprintf(fpTT, "minTotalTT,avgTotalTT,maxTotalTT\n");

}

fprintf(fp, "%.1f,", ratioInter);
fprintf(fp, "%lu,%lu,", countInter, countNoInter);
double avgInter = sumInter / (double)countInter;
fprintf(fp, "%f,%f,%f,", minInter, avgInter, maxInter);
double avgNoInter = sumNoInter / (double)countNoInter;
fprintf(fp, "%f,%f,%f,", minNoInter, avgNoInter, maxNoInter);
double avg =
    ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
fprintf(fp, "%f,%f,%f",
    (minNoInter < minInter ? minNoInter : minInter),
    avg,

```



```

        (maxNoInter > maxInter ? maxNoInter : maxInter));
if (iRun < NB_RUNS - 1) {
    fprintf(fp, "\n");
}

fprintf(fpCC, "%.1f,", ratioInter);
fprintf(fpCC, "%lu,%lu,", countInterCC, countNoInterCC);
double avgInterCC = sumInterCC / (double)countInterCC;
fprintf(fpCC, "%f,%f,%f,", minInterCC, avgInterCC, maxInterCC);
double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
fprintf(fpCC, "%f,%f,%f,", minNoInterCC, avgNoInterCC, maxNoInterCC);
double avgCC =
    ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
fprintf(fpCC, "%f,%f,%f",
    (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
    avgCC,
    (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));
if (iRun < NB_RUNS - 1) {
    fprintf(fpCC, "\n");
}

fprintf(fpCT, "%.1f,", ratioInter);
fprintf(fpCT, "%lu,%lu,", countInterCT, countNoInterCT);
double avgInterCT = sumInterCT / (double)countInterCT;
fprintf(fpCT, "%f,%f,%f,", minInterCT, avgInterCT, maxInterCT);
double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
fprintf(fpCT, "%f,%f,%f,", minNoInterCT, avgNoInterCT, maxNoInterCT);
double avgCT =
    ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
fprintf(fpCT, "%f,%f,%f",
    (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
    avgCT,
    (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));
if (iRun < NB_RUNS - 1) {
    fprintf(fpCT, "\n");
}

fprintf(fpTC, "%.1f,", ratioInter);
fprintf(fpTC, "%lu,%lu,", countInterTC, countNoInterTC);
double avgInterTC = sumInterTC / (double)countInterTC;
fprintf(fpTC, "%f,%f,%f,", minInterTC, avgInterTC, maxInterTC);
double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
fprintf(fpTC, "%f,%f,%f,", minNoInterTC, avgNoInterTC, maxNoInterTC);
double avgTC =
    ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
fprintf(fpTC, "%f,%f,%f",
    (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
    avgTC,
    (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));
if (iRun < NB_RUNS - 1) {
    fprintf(fpTC, "\n");
}

fprintf(fpTT, "%.1f,", ratioInter);
fprintf(fpTT, "%lu,%lu,", countInterTT, countNoInterTT);
double avgInterTT = sumInterTT / (double)countInterTT;
fprintf(fpTT, "%f,%f,%f,", minInterTT, avgInterTT, maxInterTT);
double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
fprintf(fpTT, "%f,%f,%f,", minNoInterTT, avgNoInterTT, maxNoInterTT);
double avgTT =
    ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
fprintf(fpTT, "%f,%f,%f",

```

```

        (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
        avgTT,
        (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));
    if (iRun < NB_RUNS - 1) {
        fprintf(fpTT, "\n");
    }
}

// Close the files
fclose(fp);
fclose(fpCC);
fclose(fpCT);
fclose(fpTC);
fclose(fpTT);
}

int main(int argc, char** argv) {

    Qualify2DStatic();

    return 0;
}

```

8.1.2 3D static

```

// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>

// Include FMB and SAT algorithm library
#include "fmb3d.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 2000
// 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
    srand(time(NULL));

    // Open the files to save the results
    FILE* fp = fopen("../Results/qualification3D.txt", "w");
    FILE* fpCC = fopen("../Results/qualification3DCC.txt", "w");
    FILE* fpCT = fopen("../Results/qualification3DCT.txt", "w");
    FILE* fpTC = fopen("../Results/qualification3DTC.txt", "w");
    FILE* fpTT = fopen("../Results/qualification3DTT.txt", "w");

    // Loop on runs
    for (int iRun = 0;
        iRun < NB_RUNS;
        ++iRun) {

        // Ratio intersection/no intersection for the displayed results
        double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

        // Initialize counters
        minInter = 0.0;
        maxInter = 0.0;
        sumInter = 0.0;
        countInter = 0;
        minNoInter = 0.0;
        maxNoInter = 0.0;
        sumNoInter = 0.0;
        countNoInter = 0;

        minInterCC = 0.0;
        maxInterCC = 0.0;
        sumInterCC = 0.0;

```

```

countInterCC = 0;
minNoInterCC = 0.0;
maxNoInterCC = 0.0;
sumNoInterCC = 0.0;
countNoInterCC = 0;

minInterCT = 0.0;
maxInterCT = 0.0;
sumInterCT = 0.0;
countInterCT = 0;
minNoInterCT = 0.0;
maxNoInterCT = 0.0;
sumNoInterCT = 0.0;
countNoInterCT = 0;

minInterTC = 0.0;
maxInterTC = 0.0;
sumInterTC = 0.0;
countInterTC = 0;
minNoInterTC = 0.0;
maxNoInterTC = 0.0;
sumNoInterTC = 0.0;
countNoInterTC = 0;

minInterTT = 0.0;
maxInterTT = 0.0;
sumInterTT = 0.0;
countInterTT = 0;
minNoInterTT = 0.0;
maxNoInterTT = 0.0;
sumNoInterTT = 0.0;
countNoInterTT = 0;

// Declare two variables to memorize the arguments to the
// Qualification function
Param3D paramP;
Param3D paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

    // Create two random Frame definitions
    Param3D* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

        // 50% chance of being a Cuboid or a Tetrahedron
        if (rnd() < 0.5)
            param->type = FrameCuboid;
        else
            param->type = FrameTetrahedron;

        for (int iAxis = 3;
             iAxis--;) {

            param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            for (int iComp = 3;
                 iComp--;) {

                param->comp[iComp][iAxis] =

```

```

        -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

    }

}

param = &paramQ;

}

// Calculate the determinant of the Frames' components matrix
double detP =
    paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2] -
    paramP.comp[1][2] * paramP.comp[2][1]) -
    paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2] -
    paramP.comp[0][2] * paramP.comp[2][1]) +
    paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2] -
    paramP.comp[0][2] * paramP.comp[1][1]);

double detQ =
    paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2] -
    paramQ.comp[1][2] * paramQ.comp[2][1]) -
    paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2] -
    paramQ.comp[0][2] * paramQ.comp[2][1]) +
    paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2] -
    paramQ.comp[0][2] * paramQ.comp[1][1]);

// If the determinants are not null, ie the Frame are not degenerate
if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Qualification3DStatic(
        paramP,
        paramQ);

}

}

// Display the results
if (iRun == 0) {

    fprintf(fp, "percPairInter,");
    fprintf(fp, "countInterTo,countNoInterTo,");
    fprintf(fp, "minInterTo,avgInterTo,maxInterTo,");
    fprintf(fp, "minNoInterTo,avgNoInterTo,maxNoInterTo,");
    fprintf(fp, "minTotalTo,avgTotalTo,maxTotalTo\n");

    fprintf(fpCC, "percPairInter,");
    fprintf(fpCC, "countInterCC,countNoInterCC,");
    fprintf(fpCC, "minInterCC,avgInterCC,maxInterCC,");
    fprintf(fpCC, "minNoInterCC,avgNoInterCC,maxNoInterCC,");
    fprintf(fpCC, "minTotalCC,avgTotalCC,maxTotalCC\n");

    fprintf(fpCT, "percPairInter,");
    fprintf(fpCT, "countInterCT,countNoInterCT,");
    fprintf(fpCT, "minInterCT,avgInterCT,maxInterCT,");
    fprintf(fpCT, "minNoInterCT,avgNoInterCT,maxNoInterCT,");
    fprintf(fpCT, "minTotalCT,avgTotalCT,maxTotalCT\n");

    fprintf(fpTC, "percPairInter,");
    fprintf(fpTC, "countInterTC,countNoInterTC,");

```

```

    fprintf(fpTC, "minInterTC,avgInterTC,maxInterTC,");
    fprintf(fpTC, "minNoInterTC,avgNoInterTC,maxNoInterTC,");
    fprintf(fpTC, "minTotalTC,avgTotalTC,maxTotalTC\n");

    fprintf(fpTT, "percPairInter,");
    fprintf(fpTT, "countInterTT,countNoInterTT,");
    fprintf(fpTT, "minInterTT,avgInterTT,maxInterTT,");
    fprintf(fpTT, "minNoInterTT,avgNoInterTT,maxNoInterTT,");
    fprintf(fpTT, "minTotalTT,avgTotalTT,maxTotalTT\n");
}

fprintf(fp, "%.1f,", ratioInter);
fprintf(fp, "%lu,%lu,", countInter, countNoInter);
double avgInter = sumInter / (double)countInter;
fprintf(fp, "%f,%f,%f,", minInter, avgInter, maxInter);
double avgNoInter = sumNoInter / (double)countNoInter;
fprintf(fp, "%f,%f,%f,", minNoInter, avgNoInter, maxNoInter);
double avg =
    ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
fprintf(fp, "%f,%f,%f",
    (minNoInter < minInter ? minNoInter : minInter),
    avg,
    (maxNoInter > maxInter ? maxNoInter : maxInter));
if (iRun < NB_RUNS - 1) {
    fprintf(fp, "\n");
}

fprintf(fpCC, "%.1f,", ratioInter);
fprintf(fpCC, "%lu,%lu,", countInterCC, countNoInterCC);
double avgInterCC = sumInterCC / (double)countInterCC;
fprintf(fpCC, "%f,%f,%f,", minInterCC, avgInterCC, maxInterCC);
double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
fprintf(fpCC, "%f,%f,%f,", minNoInterCC, avgNoInterCC, maxNoInterCC);
double avgCC =
    ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
fprintf(fpCC, "%f,%f,%f",
    (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
    avgCC,
    (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));
if (iRun < NB_RUNS - 1) {
    fprintf(fpCC, "\n");
}

fprintf(fpCT, "%.1f,", ratioInter);
fprintf(fpCT, "%lu,%lu,", countInterCT, countNoInterCT);
double avgInterCT = sumInterCT / (double)countInterCT;
fprintf(fpCT, "%f,%f,%f,", minInterCT, avgInterCT, maxInterCT);
double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
fprintf(fpCT, "%f,%f,%f,", minNoInterCT, avgNoInterCT, maxNoInterCT);
double avgCT =
    ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
fprintf(fpCT, "%f,%f,%f",
    (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
    avgCT,
    (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));
if (iRun < NB_RUNS - 1) {
    fprintf(fpCT, "\n");
}

fprintf(fpTC, "%.1f,", ratioInter);
fprintf(fpTC, "%lu,%lu,", countInterTC, countNoInterTC);

```

```

double avgInterTC = sumInterTC / (double)countInterTC;
fprintf(fpTC, "%f,%f,%f,", minInterTC, avgInterTC, maxInterTC);
double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
fprintf(fpTC, "%f,%f,%f,", minNoInterTC, avgNoInterTC, maxNoInterTC);
double avgTC =
    ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
fprintf(fpTC, "%f,%f,%f",
    (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
    avgTC,
    (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));
if (iRun < NB_RUNS - 1) {
    fprintf(fpTC, "\n");
}

fprintf(fpTT, "%.1f,", ratioInter);
fprintf(fpTT, "%lu,%lu,", countInterTT, countNoInterTT);
double avgInterTT = sumInterTT / (double)countInterTT;
fprintf(fpTT, "%f,%f,%f,", minInterTT, avgInterTT, maxInterTT);
double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
fprintf(fpTT, "%f,%f,%f,", minNoInterTT, avgNoInterTT, maxNoInterTT);
double avgTT =
    ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
fprintf(fpTT, "%f,%f,%f",
    (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
    avgTT,
    (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));
if (iRun < NB_RUNS - 1) {
    fprintf(fpTT, "\n");
}

}

// Close the files
fclose(fp);
fclose(fpCC);
fclose(fpCT);
fclose(fpTC);
fclose(fpTT);

}

int main(int argc, char** argv) {

    Qualify3DStatic();

    return 0;
}

```

8.1.3 2D dynamic

```

// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>

// Include FMB and SAT algorithm library
#include "fmb2dt.h"
#include "sat.h"

```

```

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 2000
// 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
    srand(time(NULL));

    // Open the files to save the results
    FILE* fp = fopen("../Results/qualification2DTime.txt", "w");

```

```

FILE* fpCC = fopen("../Results/qualification2DTimeCC.txt", "w");
FILE* fpCT = fopen("../Results/qualification2DTimeCT.txt", "w");
FILE* fpTC = fopen("../Results/qualification2DTimeTC.txt", "w");
FILE* fpTT = fopen("../Results/qualification2DTimeTT.txt", "w");

// Loop on runs
for (int iRun = 0;
     iRun < NB_RUNS;
     ++iRun) {

    // Ratio intersection/no intersection for the displayed results
    double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

    // Initialize counters
    minInter = 0.0;
    maxInter = 0.0;
    sumInter = 0.0;
    countInter = 0;
    minNoInter = 0.0;
    maxNoInter = 0.0;
    sumNoInter = 0.0;
    countNoInter = 0;

    minInterCC = 0.0;
    maxInterCC = 0.0;
    sumInterCC = 0.0;
    countInterCC = 0;
    minNoInterCC = 0.0;
    maxNoInterCC = 0.0;
    sumNoInterCC = 0.0;
    countNoInterCC = 0;

    minInterCT = 0.0;
    maxInterCT = 0.0;
    sumInterCT = 0.0;
    countInterCT = 0;
    minNoInterCT = 0.0;
    maxNoInterCT = 0.0;
    sumNoInterCT = 0.0;
    countNoInterCT = 0;

    minInterTC = 0.0;
    maxInterTC = 0.0;
    sumInterTC = 0.0;
    countInterTC = 0;
    minNoInterTC = 0.0;
    maxNoInterTC = 0.0;
    sumNoInterTC = 0.0;
    countNoInterTC = 0;

    minInterTT = 0.0;
    maxInterTT = 0.0;
    sumInterTT = 0.0;
    countInterTT = 0;
    minNoInterTT = 0.0;
    maxNoInterTT = 0.0;
    sumNoInterTT = 0.0;
    countNoInterTT = 0;

    // Declare two variables to memorize the arguments to the
    // Qualification function
    Param2DTime paramP;

```

```

Param2DTime paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

    // Create two random Frame definitions
    Param2DTime* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

        // 50% chance of being a Cuboid or a Tetrahedron
        if (rnd() < 0.5)
            param->type = FrameCuboid;
        else
            param->type = FrameTetrahedron;

        for (int iAxis = 2;
             iAxis--;) {

            param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
            param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            for (int iComp = 2;
                 iComp--;) {

                param->comp[iComp][iAxis] =
                    -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            }

        }

        param = &paramQ;
    }

    // Calculate the determinant of the Frames' components matrix

    double detP =
        paramP.comp[0][0] * paramP.comp[1][1] -
        paramP.comp[1][0] * paramP.comp[0][1];

    double detQ =
        paramQ.comp[0][0] * paramQ.comp[1][1] -
        paramQ.comp[1][0] * paramQ.comp[0][1];

    // If the determinants are not null, ie the Frame are not degenerate
    if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

        // Run the validation on the two Frames
        Qualification2DDynamic(
            paramP,
            paramQ);

    }

}

// Display the results
if (iRun == 0) {

```

```

fprintf(fp, "percPairInter,");
fprintf(fp, "countInterTo,countNoInterTo,");
fprintf(fp, "minInterTo,avgInterTo,maxInterTo,");
fprintf(fp, "minNoInterTo,avgNoInterTo,maxNoInterTo,");
fprintf(fp, "minTotalTo,avgTotalTo,maxTotalTo\n");

fprintf(fpCC, "percPairInter,");
fprintf(fpCC, "countInterCC,countNoInterCC,");
fprintf(fpCC, "minInterCC,avgInterCC,maxInterCC,");
fprintf(fpCC, "minNoInterCC,avgNoInterCC,maxNoInterCC,");
fprintf(fpCC, "minTotalCC,avgTotalCC,maxTotalCC\n");

fprintf(fpCT, "percPairInter,");
fprintf(fpCT, "countInterCT,countNoInterCT,");
fprintf(fpCT, "minInterCT,avgInterCT,maxInterCT,");
fprintf(fpCT, "minNoInterCT,avgNoInterCT,maxNoInterCT,");
fprintf(fpCT, "minTotalCT,avgTotalCT,maxTotalCT\n");

fprintf(fpTC, "percPairInter,");
fprintf(fpTC, "countInterTC,countNoInterTC,");
fprintf(fpTC, "minInterTC,avgInterTC,maxInterTC,");
fprintf(fpTC, "minNoInterTC,avgNoInterTC,maxNoInterTC,");
fprintf(fpTC, "minTotalTC,avgTotalTC,maxTotalTC\n");

fprintf(fpTT, "percPairInter,");
fprintf(fpTT, "countInterTT,countNoInterTT,");
fprintf(fpTT, "minInterTT,avgInterTT,maxInterTT,");
fprintf(fpTT, "minNoInterTT,avgNoInterTT,maxNoInterTT,");
fprintf(fpTT, "minTotalTT,avgTotalTT,maxTotalTT\n");
}

fprintf(fp, "%.1f,", ratioInter);
fprintf(fp, "%lu,%lu,", countInter, countNoInter);
double avgInter = sumInter / (double)countInter;
fprintf(fp, "%f,%f,%f,", minInter, avgInter, maxInter);
double avgNoInter = sumNoInter / (double)countNoInter;
fprintf(fp, "%f,%f,%f,", minNoInter, avgNoInter, maxNoInter);
double avg =
    ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
fprintf(fp, "%f,%f,%f",
    (minNoInter < minInter ? minNoInter : minInter),
    avg,
    (maxNoInter > maxInter ? maxNoInter : maxInter));
if (iRun < NB_RUNS - 1) {
    fprintf(fp, "\n");
}

fprintf(fpCC, "%.1f,", ratioInter);
fprintf(fpCC, "%lu,%lu,", countInterCC, countNoInterCC);
double avgInterCC = sumInterCC / (double)countInterCC;
fprintf(fpCC, "%f,%f,%f,", minInterCC, avgInterCC, maxInterCC);
double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
fprintf(fpCC, "%f,%f,%f,", minNoInterCC, avgNoInterCC, maxNoInterCC);
double avgCC =
    ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
fprintf(fpCC, "%f,%f,%f",
    (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
    avgCC,
    (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));
if (iRun < NB_RUNS - 1) {
    fprintf(fpCC, "\n");
}

```

```

    }

    fprintf(fpCT, "%.1f,", ratioInter);
    fprintf(fpCT, "%lu,%lu,", countInterCT, countNoInterCT);
    double avgInterCT = sumInterCT / (double)countInterCT;
    fprintf(fpCT, "%f,%f,%f,", minInterCT, avgInterCT, maxInterCT);
    double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
    fprintf(fpCT, "%f,%f,%f,", minNoInterCT, avgNoInterCT, maxNoInterCT);
    double avgCT =
        ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
    fprintf(fpCT, "%f,%f,%f",
        (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
        avgCT,
        (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));
    if (iRun < NB_RUNS - 1) {
        fprintf(fpCT, "\n");
    }

    fprintf(fpTC, "%.1f,", ratioInter);
    fprintf(fpTC, "%lu,%lu,", countInterTC, countNoInterTC);
    double avgInterTC = sumInterTC / (double)countInterTC;
    fprintf(fpTC, "%f,%f,%f,", minInterTC, avgInterTC, maxInterTC);
    double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
    fprintf(fpTC, "%f,%f,%f,", minNoInterTC, avgNoInterTC, maxNoInterTC);
    double avgTC =
        ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
    fprintf(fpTC, "%f,%f,%f",
        (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
        avgTC,
        (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));
    if (iRun < NB_RUNS - 1) {
        fprintf(fpTC, "\n");
    }

    fprintf(fpTT, "%.1f,", ratioInter);
    fprintf(fpTT, "%lu,%lu,", countInterTT, countNoInterTT);
    double avgInterTT = sumInterTT / (double)countInterTT;
    fprintf(fpTT, "%f,%f,%f,", minInterTT, avgInterTT, maxInterTT);
    double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
    fprintf(fpTT, "%f,%f,%f,", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
        ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    fprintf(fpTT, "%f,%f,%f",
        (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
        avgTT,
        (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));
    if (iRun < NB_RUNS - 1) {
        fprintf(fpTT, "\n");
    }

}

// Close the files
fclose(fp);
fclose(fpCC);
fclose(fpCT);
fclose(fpTC);
fclose(fpTT);

}

int main(int argc, char** argv) {

```



```

    Qualify2DDynamic();

    return 0;
}

```

8.1.4 3D dynamic

```

// Include standard libraries
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>

// Include FMB and SAT algorithm library
#include "fmb3dt.h"
#include "sat.h"

// Epsilon to detect degenerated triangles
#define EPSILON 0.1
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 9
// Nb of tests per run
#define NB_TESTS 2000
// 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
    srand(time(NULL));

    // Open the files to save the results
    FILE* fp = fopen("../Results/qualification3DTime.txt", "w");
    FILE* fpCC = fopen("../Results/qualification3DTimeCC.txt", "w");
    FILE* fpCT = fopen("../Results/qualification3DTimeCT.txt", "w");
    FILE* fpTC = fopen("../Results/qualification3DTimeTC.txt", "w");
    FILE* fpTT = fopen("../Results/qualification3DTimeTT.txt", "w");

    // Loop on runs
    for (int iRun = 0;
        iRun < NB_RUNS;
        ++iRun) {

        // Ratio intersection/no intersection for the displayed results
        double ratioInter = 0.1 + 0.8 * (double)iRun / (double)(NB_RUNS - 1);

        // Initialize counters
        minInter = 0.0;
        maxInter = 0.0;
        sumInter = 0.0;
        countInter = 0;
        minNoInter = 0.0;
        maxNoInter = 0.0;
        sumNoInter = 0.0;
        countNoInter = 0;

        minInterCC = 0.0;
        maxInterCC = 0.0;
        sumInterCC = 0.0;
        countInterCC = 0;
        minNoInterCC = 0.0;
        maxNoInterCC = 0.0;
        sumNoInterCC = 0.0;
        countNoInterCC = 0;

        minInterCT = 0.0;
        maxInterCT = 0.0;
        sumInterCT = 0.0;
        countInterCT = 0;
        minNoInterCT = 0.0;
        maxNoInterCT = 0.0;
        sumNoInterCT = 0.0;
        countNoInterCT = 0;

        minInterTC = 0.0;

```



```

maxInterTC = 0.0;
sumInterTC = 0.0;
countInterTC = 0;
minNoInterTC = 0.0;
maxNoInterTC = 0.0;
sumNoInterTC = 0.0;
countNoInterTC = 0;

minInterTT = 0.0;
maxInterTT = 0.0;
sumInterTT = 0.0;
countInterTT = 0;
minNoInterTT = 0.0;
maxNoInterTT = 0.0;
sumNoInterTT = 0.0;
countNoInterTT = 0;

// Declare two variables to memorize the arguments to the
// Qualification function
Param3DTime paramP;
Param3DTime paramQ;

// Loop on the number of tests
for (unsigned long iTest = NB_TESTS;
     iTest--;) {

    // Create two random Frame definitions
    Param3DTime* param = &paramP;
    for (int iParam = 2;
         iParam--;) {

        // 50% chance of being a Cuboid or a Tetrahedron
        if (rnd() < 0.5)
            param->type = FrameCuboid;
        else
            param->type = FrameTetrahedron;

        for (int iAxis = 3;
             iAxis--;) {

            param->orig[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;
            param->speed[iAxis] = -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            for (int iComp = 3;
                 iComp--;) {

                param->comp[iComp][iAxis] =
                    -RANGE_AXIS + 2.0 * rnd() * RANGE_AXIS;

            }

        }

        param = &paramQ;

    }

    // Calculate the determinant of the Frames' components matrix
    double detP =
        paramP.comp[0][0] * (paramP.comp[1][1] * paramP.comp[2][2] -
                             paramP.comp[1][2] * paramP.comp[2][1]) -

```

```

paramP.comp[1][0] * (paramP.comp[0][1] * paramP.comp[2][2] -
paramP.comp[0][2] * paramP.comp[2][1]) +
paramP.comp[2][0] * (paramP.comp[0][1] * paramP.comp[1][2] -
paramP.comp[0][2] * paramP.comp[1][1]);

double detQ =
paramQ.comp[0][0] * (paramQ.comp[1][1] * paramQ.comp[2][2] -
paramQ.comp[1][2] * paramQ.comp[2][1]) -
paramQ.comp[1][0] * (paramQ.comp[0][1] * paramQ.comp[2][2] -
paramQ.comp[0][2] * paramQ.comp[2][1]) +
paramQ.comp[2][0] * (paramQ.comp[0][1] * paramQ.comp[1][2] -
paramQ.comp[0][2] * paramQ.comp[1][1]);

// If the determinants are not null, ie the Frame are not degenerate
if (fabs(detP) > EPSILON && fabs(detQ) > EPSILON) {

    // Run the validation on the two Frames
    Qualification3DDynamic(
        paramP,
        paramQ);

}

}

// Display the results
if (iRun == 0) {

    fprintf(fp, "percPairInter,");
    fprintf(fp, "countInterTo,countNoInterTo,");
    fprintf(fp, "minInterTo,avgInterTo,maxInterTo,");
    fprintf(fp, "minNoInterTo,avgNoInterTo,maxNoInterTo,");
    fprintf(fp, "minTotalTo,avgTotalTo,maxTotalTo\n");

    fprintf(fpCC, "percPairInter,");
    fprintf(fpCC, "countInterCC,countNoInterCC,");
    fprintf(fpCC, "minInterCC,avgInterCC,maxInterCC,");
    fprintf(fpCC, "minNoInterCC,avgNoInterCC,maxNoInterCC,");
    fprintf(fpCC, "minTotalCC,avgTotalCC,maxTotalCC\n");

    fprintf(fpCT, "percPairInter,");
    fprintf(fpCT, "countInterCT,countNoInterCT,");
    fprintf(fpCT, "minInterCT,avgInterCT,maxInterCT,");
    fprintf(fpCT, "minNoInterCT,avgNoInterCT,maxNoInterCT,");
    fprintf(fpCT, "minTotalCT,avgTotalCT,maxTotalCT\n");

    fprintf(fpTC, "percPairInter,");
    fprintf(fpTC, "countInterTC,countNoInterTC,");
    fprintf(fpTC, "minInterTC,avgInterTC,maxInterTC,");
    fprintf(fpTC, "minNoInterTC,avgNoInterTC,maxNoInterTC,");
    fprintf(fpTC, "minTotalTC,avgTotalTC,maxTotalTC\n");

    fprintf(fpTT, "percPairInter,");
    fprintf(fpTT, "countInterTT,countNoInterTT,");
    fprintf(fpTT, "minInterTT,avgInterTT,maxInterTT,");
    fprintf(fpTT, "minNoInterTT,avgNoInterTT,maxNoInterTT,");
    fprintf(fpTT, "minTotalTT,avgTotalTT,maxTotalTT\n");

}

fprintf(fp, "%.1f,", ratioInter);
fprintf(fp, "%lu,%lu,", countInter, countNoInter);

```

```

double avgInter = sumInter / (double)countInter;
fprintf(fp, "%f,%f,%f,", minInter, avgInter, maxInter);
double avgNoInter = sumNoInter / (double)countNoInter;
fprintf(fp, "%f,%f,%f,", minNoInter, avgNoInter, maxNoInter);
double avg =
    ratioInter * avgInter + (1.0 - ratioInter) * avgNoInter;
fprintf(fp, "%f,%f,%f",
    (minNoInter < minInter ? minNoInter : minInter),
    avg,
    (maxNoInter > maxInter ? maxNoInter : maxInter));
if (iRun < NB_RUNS - 1) {
    fprintf(fp, "\n");
}

fprintf(fpCC, "%.1f,", ratioInter);
fprintf(fpCC, "%lu,%lu,", countInterCC, countNoInterCC);
double avgInterCC = sumInterCC / (double)countInterCC;
fprintf(fpCC, "%f,%f,%f,", minInterCC, avgInterCC, maxInterCC);
double avgNoInterCC = sumNoInterCC / (double)countNoInterCC;
fprintf(fpCC, "%f,%f,%f,", minNoInterCC, avgNoInterCC, maxNoInterCC);
double avgCC =
    ratioInter * avgInterCC + (1.0 - ratioInter) * avgNoInterCC;
fprintf(fpCC, "%f,%f,%f",
    (minNoInterCC < minInterCC ? minNoInterCC : minInterCC),
    avgCC,
    (maxNoInterCC > maxInterCC ? maxNoInterCC : maxInterCC));
if (iRun < NB_RUNS - 1) {
    fprintf(fpCC, "\n");
}

fprintf(fpCT, "%.1f,", ratioInter);
fprintf(fpCT, "%lu,%lu,", countInterCT, countNoInterCT);
double avgInterCT = sumInterCT / (double)countInterCT;
fprintf(fpCT, "%f,%f,%f,", minInterCT, avgInterCT, maxInterCT);
double avgNoInterCT = sumNoInterCT / (double)countNoInterCT;
fprintf(fpCT, "%f,%f,%f,", minNoInterCT, avgNoInterCT, maxNoInterCT);
double avgCT =
    ratioInter * avgInterCT + (1.0 - ratioInter) * avgNoInterCT;
fprintf(fpCT, "%f,%f,%f",
    (minNoInterCT < minInterCT ? minNoInterCT : minInterCT),
    avgCT,
    (maxNoInterCT > maxInterCT ? maxNoInterCT : maxInterCT));
if (iRun < NB_RUNS - 1) {
    fprintf(fpCT, "\n");
}

fprintf(fpTC, "%.1f,", ratioInter);
fprintf(fpTC, "%lu,%lu,", countInterTC, countNoInterTC);
double avgInterTC = sumInterTC / (double)countInterTC;
fprintf(fpTC, "%f,%f,%f,", minInterTC, avgInterTC, maxInterTC);
double avgNoInterTC = sumNoInterTC / (double)countNoInterTC;
fprintf(fpTC, "%f,%f,%f,", minNoInterTC, avgNoInterTC, maxNoInterTC);
double avgTC =
    ratioInter * avgInterTC + (1.0 - ratioInter) * avgNoInterTC;
fprintf(fpTC, "%f,%f,%f",
    (minNoInterTC < minInterTC ? minNoInterTC : minInterTC),
    avgTC,
    (maxNoInterTC > maxInterTC ? maxNoInterTC : maxInterTC));
if (iRun < NB_RUNS - 1) {
    fprintf(fpTC, "\n");
}

```

```

    fprintf(fpTT, "%.1f,", ratioInter);
    fprintf(fpTT, "%lu,%lu,", countInterTT, countNoInterTT);
    double avgInterTT = sumInterTT / (double)countInterTT;
    fprintf(fpTT, "%f,%f,%f,", minInterTT, avgInterTT, maxInterTT);
    double avgNoInterTT = sumNoInterTT / (double)countNoInterTT;
    fprintf(fpTT, "%f,%f,%f,", minNoInterTT, avgNoInterTT, maxNoInterTT);
    double avgTT =
        ratioInter * avgInterTT + (1.0 - ratioInter) * avgNoInterTT;
    fprintf(fpTT, "%f,%f,%f",
        (minNoInterTT < minInterTT ? minNoInterTT : minInterTT),
        avgTT,
        (maxNoInterTT > maxInterTT ? maxNoInterTT : maxInterTT));
    if (iRun < NB_RUNS - 1) {
        fprintf(fpTT, "\n");
    }
}

// Close the files
fclose(fp);
fclose(fpCC);
fclose(fpCT);
fclose(fpTC);
fclose(fpTT);

}

int main(int argc, char** argv) {

    Qualify3DDynamic();

    return 0;
}

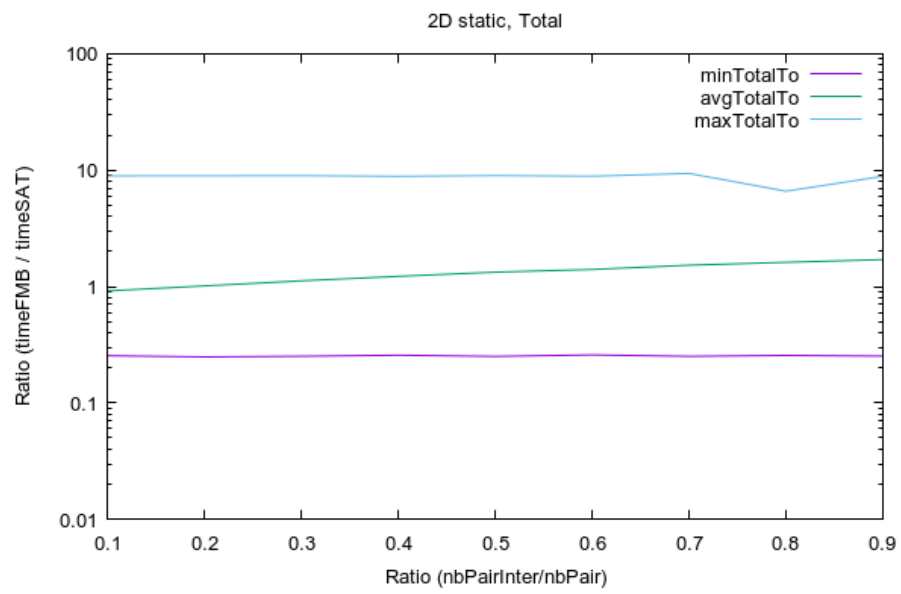
```

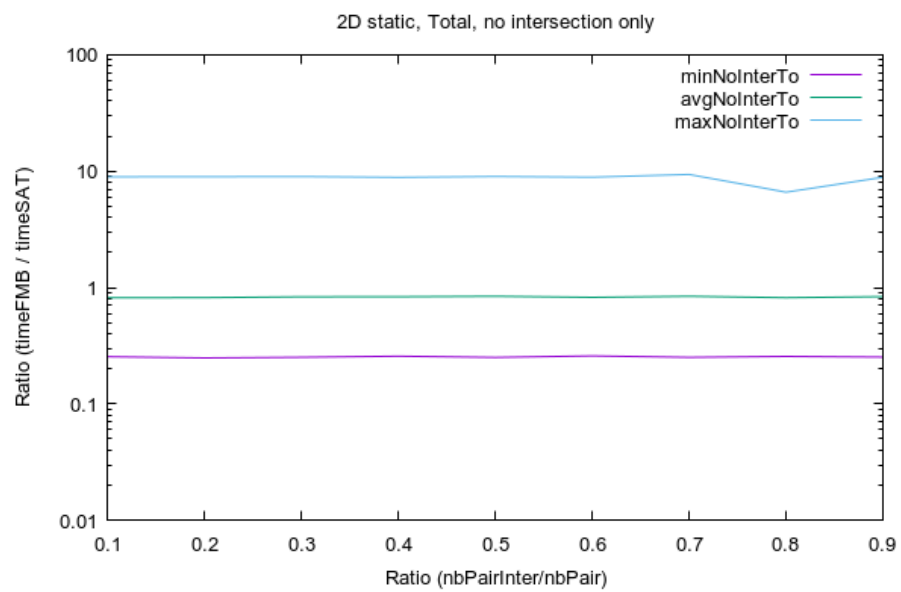
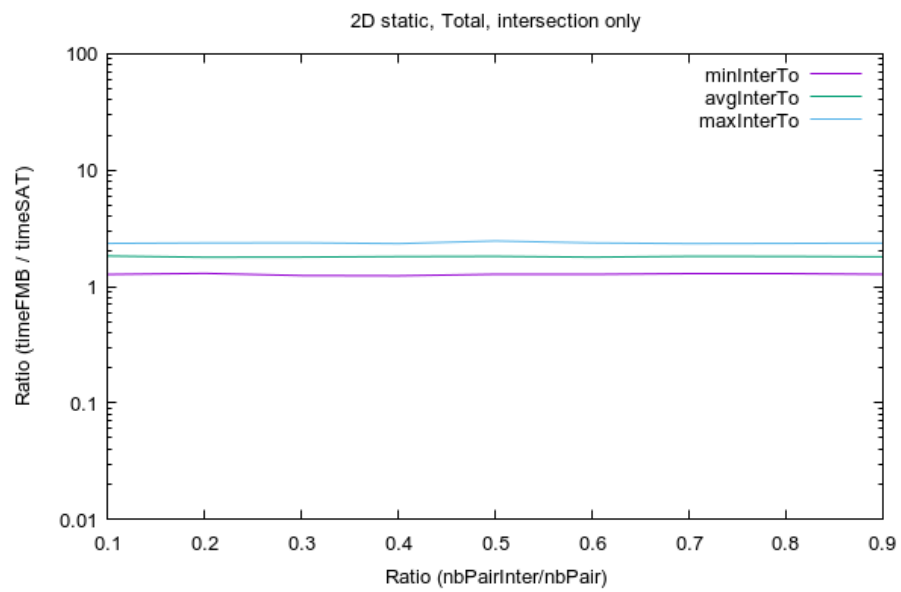
8.2 Results

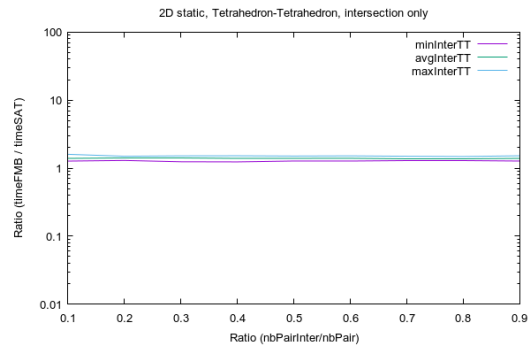
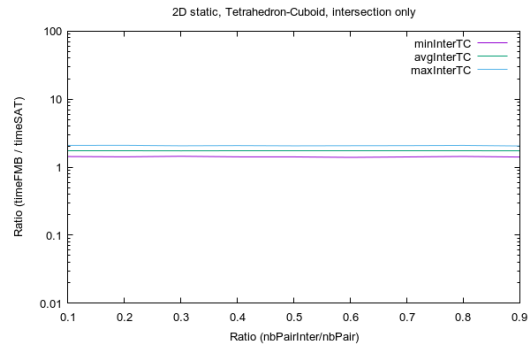
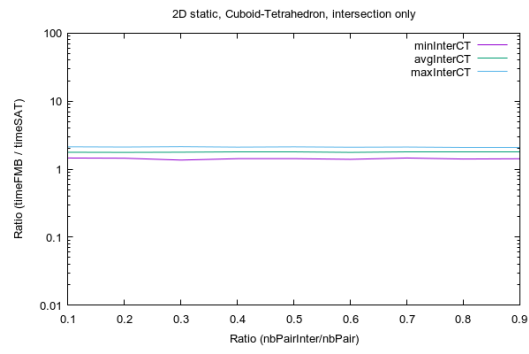
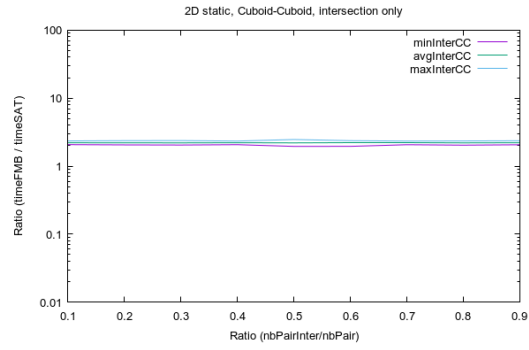
In this subsection I give the results of the qualification for each case. These results are commented in the next section.

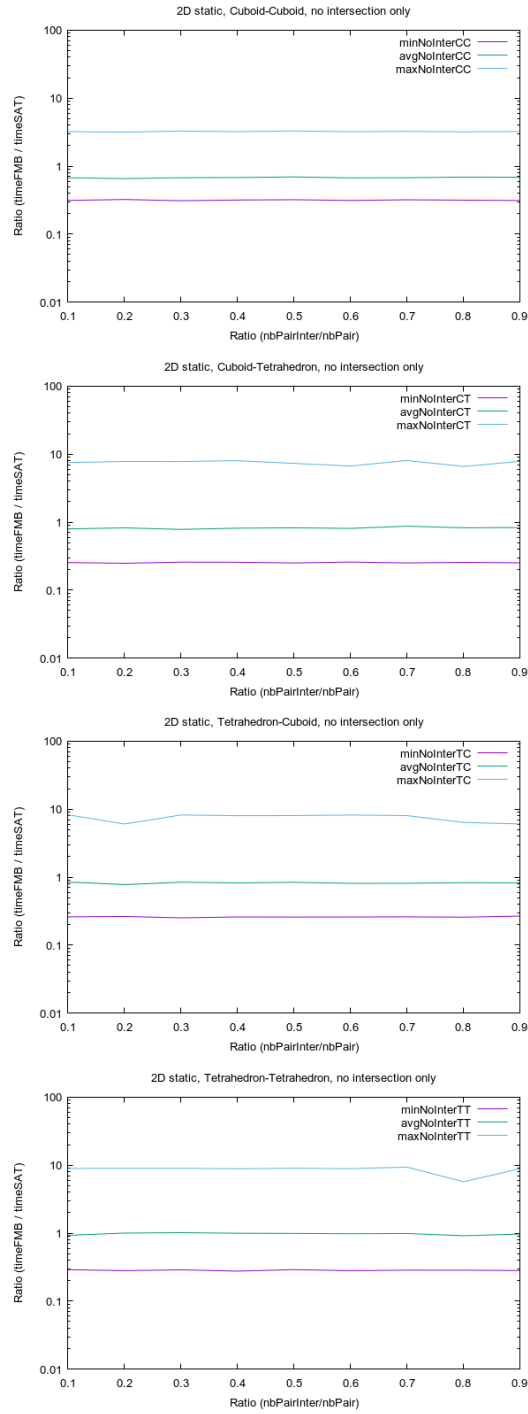
8.2.1 2D static

perPairInter	countInterTo	countNoInterTo	minInterTo	avgInterTo	maxInterTo	minNoInterTo	avgNoInterTo	maxNoInterTo	minTotalTo	avgTotalTo	maxTotalTo
0.1	966	3014	1.271902	1.830098	2.350362	0.264464	0.816691	8.949640	0.264464	0.918032	8.949640
0.2	946	3062	1.297896	2.369112	2.369112	0.248602	8.950980	8.950980	0.248602	1.013071	8.950980
0.3	1004	2996	1.242163	1.790795	2.377471	0.261771	0.833444	8.985258	0.261771	1.120650	8.985258
0.4	926	3074	1.237743	1.810022	2.338448	0.257069	0.845651	8.849412	0.257069	1.224739	8.849412
0.5	884	3116	1.275184	1.817402	2.466382	0.261101	0.841577	8.992788	0.261101	1.329489	8.992788
0.6	978	3022	1.272849	1.767699	2.369714	0.268732	0.823865	8.877150	0.268732	1.402166	8.877150
0.7	878	3122	1.276067	1.818337	2.338922	0.261418	0.841765	9.363409	0.261418	1.525365	9.363409
0.8	968	3042	1.275862	1.815645	2.349660	0.256082	0.817887	6.959822	0.256082	1.616093	6.959822
0.9	878	3122	1.272759	1.802252	2.362061	0.262845	0.835206	8.860636	0.262845	1.705548	8.860636
perPairInter	countInterCC	countNoInterCC	minInterCC	avgInterCC	maxInterCC	minNoInterCC	avgNoInterCC	maxNoInterCC	minTotalCC	avgTotalCC	maxTotalCC
0.1	296	680	2.074909	2.207525	2.350362	0.313480	0.675157	3.217064	0.313480	0.828393	3.217064
0.2	228	726	2.054844	2.203240	2.369112	0.323601	0.656291	3.168871	0.323601	0.965681	3.168871
0.3	282	706	2.047314	2.198216	2.377471	0.310318	0.676219	3.285525	0.310318	1.132818	3.285525
0.4	260	756	2.072639	2.208111	2.338448	0.317965	0.680749	3.220423	0.317965	1.291694	3.220423
0.5	268	730	1.965307	2.198240	2.466382	0.321285	0.694770	3.296061	0.321285	1.446505	3.296061
0.6	264	694	1.966635	2.211084	2.369714	0.313187	0.672102	3.212346	0.313187	1.598491	3.212346
0.7	270	726	2.069702	2.211087	2.338922	0.319968	0.676779	3.247788	0.319968	1.750795	3.247788
0.8	280	728	2.038446	2.200492	2.349660	0.316416	0.691128	3.197139	0.316416	1.896819	3.197139
0.9	238	714	2.065699	2.208362	2.362061	0.312744	0.687857	3.239375	0.312744	2.056312	3.239375
perPairInter	countInterCT	countNoInterCT	minInterCT	avgInterCT	maxInterCT	minNoInterCT	avgNoInterCT	maxNoInterCT	minTotalCT	avgTotalCT	maxTotalCT
0.1	236	774	1.459761	1.773177	2.131512	0.264464	0.795512	7.524178	0.264464	0.893278	7.524178
0.2	260	710	1.446887	1.765418	2.115954	0.248602	0.824324	7.802846	0.248602	1.012543	7.802846
0.3	254	742	1.368722	1.776264	2.140163	0.257898	0.781684	7.794661	0.257898	1.080058	7.794661
0.4	244	762	1.425940	1.773325	2.109715	0.257069	0.815620	8.014433	0.257069	1.198702	8.014433
0.5	210	802	1.427765	1.772338	2.127525	0.261101	0.825316	7.314066	0.261101	1.298827	7.314066
0.6	248	796	1.394143	1.764082	2.101455	0.268732	0.809968	6.689278	0.268732	1.382436	6.689278
0.7	202	790	1.458846	1.776337	2.14539	0.251418	0.872466	8.070213	0.251418	1.504666	8.070213
0.8	226	716	1.410037	1.776445	2.101425	0.256082	0.826430	6.595982	0.256082	1.586442	6.595982
0.9	234	794	1.421289	1.774828	2.102941	0.262845	0.833497	7.819106	0.262845	1.680695	7.819106
perPairInter	countInterTC	countNoInterTC	minInterTC	avgInterTC	maxInterTC	minNoInterTC	avgNoInterTC	maxNoInterTC	minTotalTC	avgTotalTC	maxTotalTC
0.1	288	774	1.432066	1.740505	2.089706	0.260667	0.849590	8.245763	0.260667	0.938682	8.245763
0.2	258	796	1.417349	1.742052	2.096600	0.264368	0.776926	6.055215	0.264368	0.969961	6.055215
0.3	216	768	1.445747	1.739179	2.063141	0.261771	0.842711	8.206751	0.261771	1.111652	8.206751
0.4	232	704	1.418954	1.744083	2.077970	0.269801	0.823186	8.014768	0.269801	1.191545	8.014768
0.5	228	772	1.415893	1.740609	2.059329	0.268769	0.840618	8.037267	0.268769	1.289613	8.037267
0.6	222	722	1.392186	1.740274	2.069901	0.269456	0.810589	8.205184	0.269456	1.368400	8.205184
0.7	218	798	1.414661	1.740909	2.073693	0.260841	0.812907	8.043121	0.260841	1.462508	8.043121
0.8	264	818	1.436099	1.743480	2.088999	0.267895	0.829214	6.372917	0.267895	1.560627	6.372917
0.9	220	768	1.408654	1.738661	2.050286	0.267442	0.826076	6.065606	0.267442	1.647403	6.065606
perPairInter	countInterTT	countNoInterTT	minInterTT	avgInterTT	maxInterTT	minNoInterTT	avgNoInterTT	maxNoInterTT	minTotalTT	avgTotalTT	maxTotalTT
0.1	166	786	1.271902	1.393460	1.598431	0.290487	0.927598	8.949640	0.290487	0.974184	8.949640
0.2	200	820	1.297896	1.497030	1.820505	0.280505	1.002349	8.950980	0.280505	1.080703	8.950980
0.3	252	780	1.242163	1.393763	1.516142	0.288600	0.915868	8.985258	0.288600	1.129236	8.985258
0.4	190	832	1.237743	1.382912	1.520968	0.276316	0.997345	8.849412	0.276316	1.155672	8.849412
0.5	178	812	1.275184	1.395536	1.508624	0.290909	0.990529	8.992788	0.290909	1.193033	8.992788
0.6	244	810	1.272549	1.366765	1.516348	0.280112	0.929181	8.877150	0.280112	1.23813	8.877150
0.7	188	808	1.276067	1.389944	1.496615	0.285510	0.988490	9.363409	0.285510	1.269508	9.363409
0.8	188	780	1.275862	1.390927	1.483412	0.285206	0.916476	5.702638	0.285206	1.296037	5.702638
0.9	186	846	1.272759	1.392322	1.521217	0.281780	0.969457	8.860636	0.281780	1.350036	8.860636



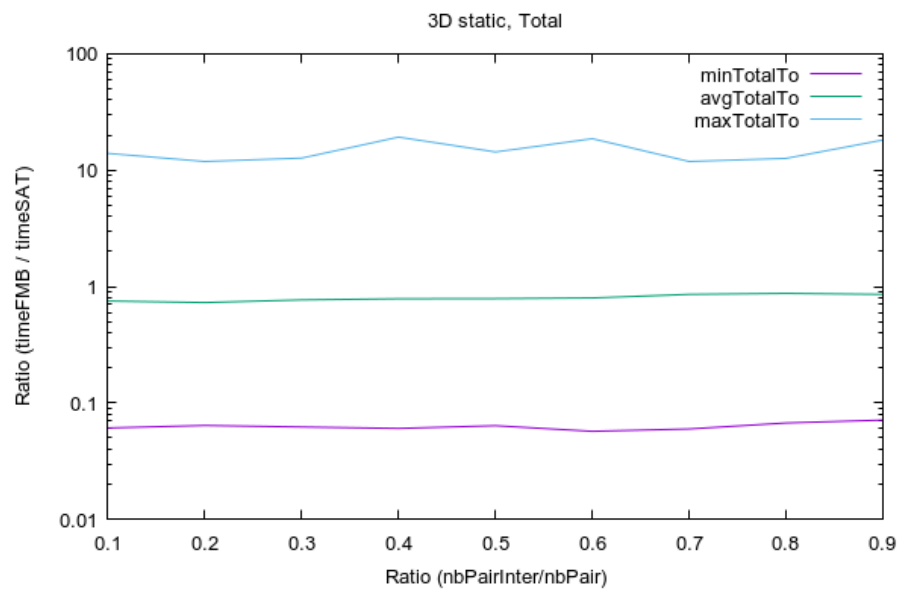


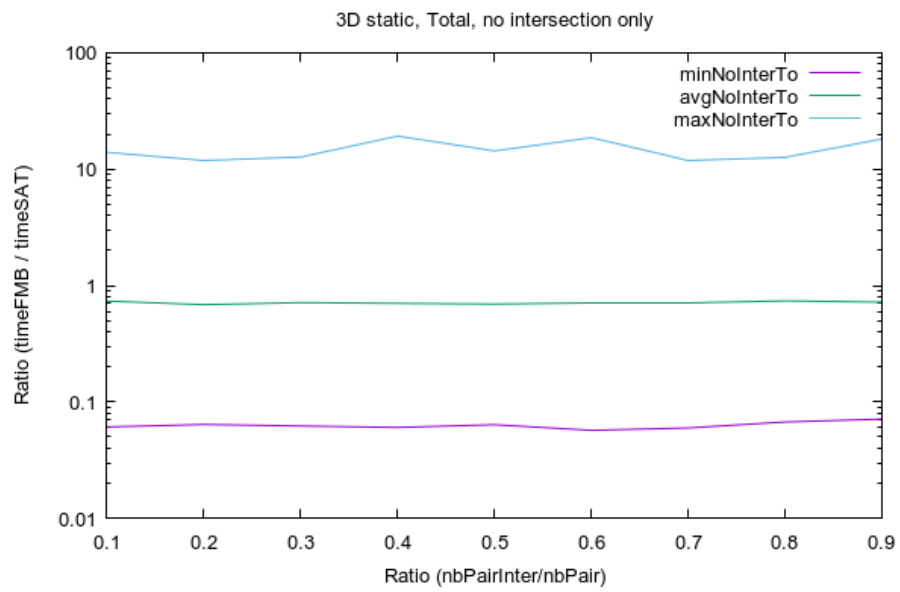
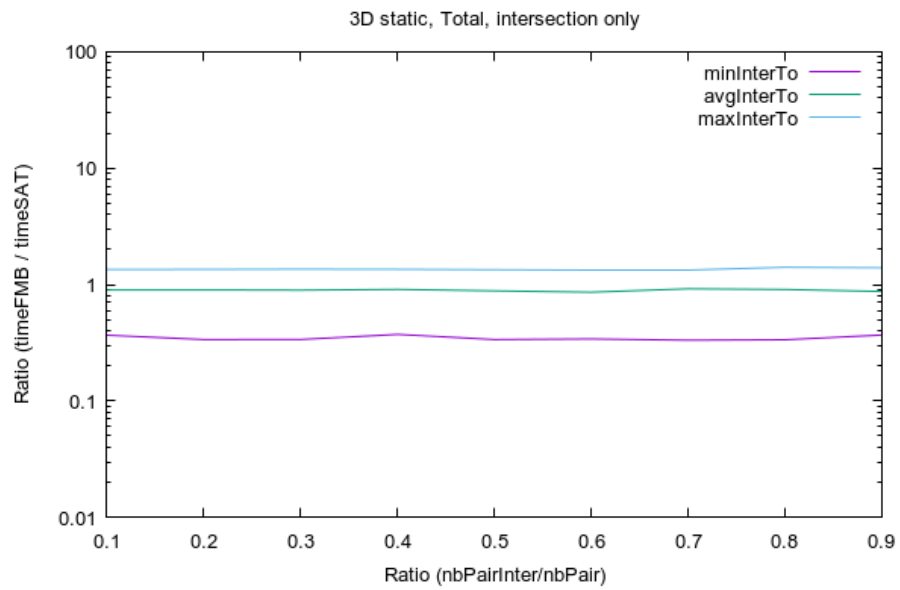


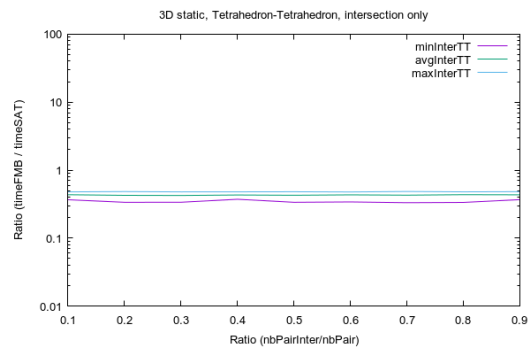
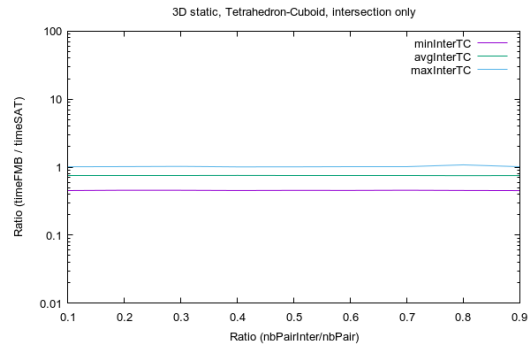
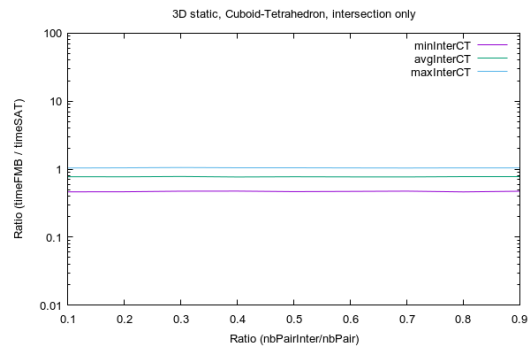
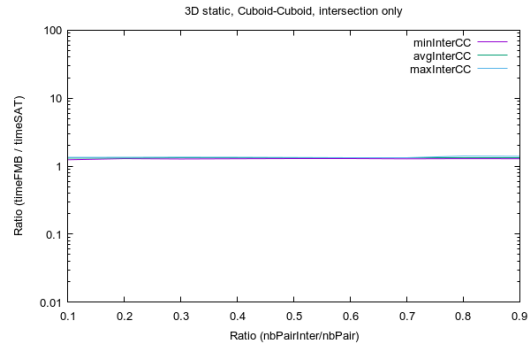


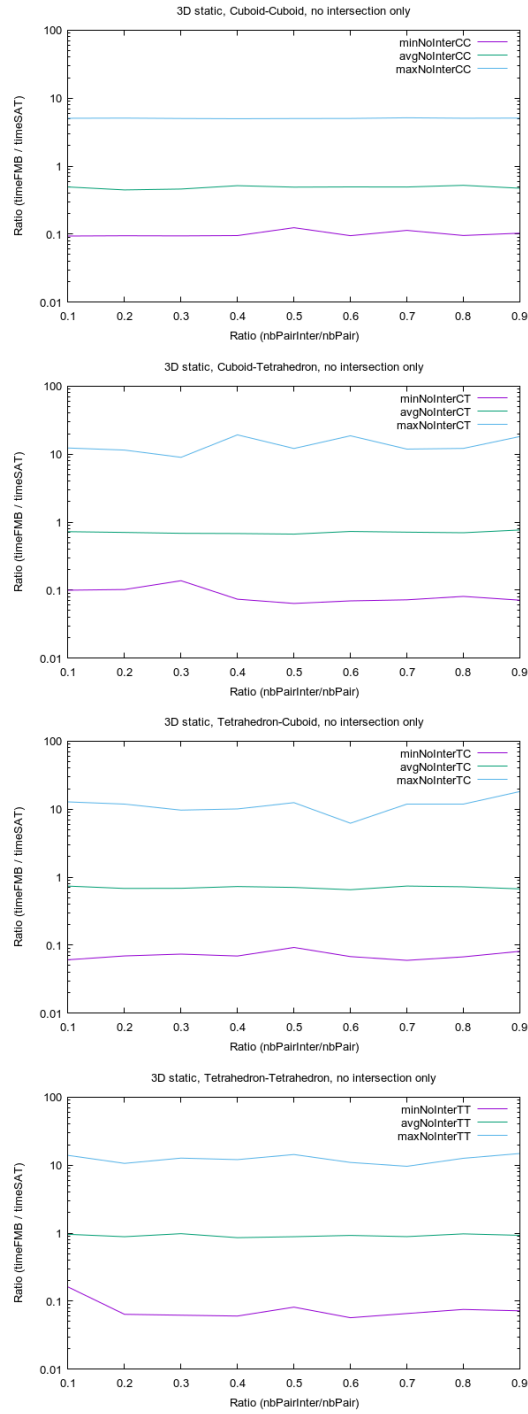
8.2.2 3D static

perPairInter	countInterTo	countNoInterTo	minInterTo	avgInterTo	maxInterTo	minNoInterTo	avgNoInterTo	maxNoInterTo	minTotalTo	avgTotalTo	maxTotalTo
0.1	624	3376	0.367096	0.897595	1.348065	0.060946	0.735422	13.943307	0.060946	0.751639	13.943307
0.2	698	3302	0.336772	0.899539	1.350218	0.064063	0.685773	11.879581	0.064063	0.728526	11.879581
0.3	690	3310	0.337496	0.895417	1.355640	0.062131	0.714007	12.704000	0.062131	0.768430	12.704000
0.4	590	3440	0.372910	0.908006	1.351076	0.060422	0.703153	19.245283	0.060422	0.785094	19.245283
0.5	666	3334	0.336753	0.882904	1.341736	0.063713	0.693462	14.341346	0.063713	0.788183	14.341346
0.6	644	3386	0.341038	0.860119	1.348163	0.057055	0.707943	18.660538	0.057055	0.799249	18.660538
0.7	624	3376	0.332448	0.920179	1.346662	0.059744	0.708817	11.865333	0.059744	0.856770	11.865333
0.8	676	3324	0.335461	0.907536	1.408037	0.067298	0.738470	12.620192	0.067298	0.873722	12.620192
0.9	588	3412	0.368328	0.872159	1.395619	0.071209	0.720603	18.168643	0.071209	0.857003	18.168643
perPairInter	countInterCC	countNoInterCC	minInterCC	avgInterCC	maxInterCC	minNoInterCC	avgNoInterCC	maxNoInterCC	minTotalCC	avgTotalCC	maxTotalCC
0.1	210	840	1.243151	1.316226	1.348065	0.093896	0.494787	5.062941	0.093896	0.576931	5.062941
0.2	260	800	1.290912	1.316936	1.350218	0.094726	0.447466	5.092896	0.094726	0.621360	5.092896
0.3	234	754	1.274288	1.315716	1.355640	0.094229	0.460967	5.020760	0.094229	0.717392	5.020760
0.4	212	778	1.284386	1.316014	1.351076	0.095255	0.516562	4.986078	0.095255	0.836397	4.986078
0.5	222	790	1.287772	1.315305	1.341736	0.124396	0.491607	5.013459	0.124396	0.903456	5.013459
0.6	196	800	1.289117	1.316417	1.348163	0.094975	0.494938	5.032982	0.094975	0.987825	5.032982
0.7	240	824	1.280064	1.315854	1.346662	0.113639	0.494050	5.142818	0.113639	1.069313	5.142818
0.8	230	762	1.315596	1.315596	1.408037	0.095523	0.522460	5.065385	0.095523	1.156969	5.065385
0.9	178	756	1.283514	1.315464	1.395619	0.103327	0.473714	5.092676	0.103327	1.231289	5.092676
perPairInter	countInterCT	countNoInterCT	minInterCT	avgInterCT	maxInterCT	minNoInterCT	avgNoInterCT	maxNoInterCT	minTotalCT	avgTotalCT	maxTotalCT
0.1	144	786	0.463621	0.774371	1.039054	0.099836	0.724941	12.329715	0.099836	0.729884	12.329715
0.2	148	788	0.465232	0.771070	1.046495	0.102126	0.706224	11.484768	0.102126	0.719194	11.484768
0.3	180	826	0.474473	0.779586	1.056411	0.137656	0.685562	8.980769	0.137656	0.713769	8.980769
0.4	130	892	0.475826	0.767265	1.040931	0.073610	0.680794	19.245283	0.073610	0.715382	19.245283
0.5	182	904	0.466780	0.773336	1.041145	0.063713	0.667545	12.130435	0.063713	0.720440	12.130435
0.6	160	832	0.470555	0.768909	1.042937	0.069511	0.729671	18.650538	0.069511	0.753153	18.650538
0.7	144	910	0.475023	0.768702	1.039750	0.072063	0.712598	11.865333	0.072063	0.751871	11.865333
0.8	178	804	0.462735	0.772779	1.044820	0.080906	0.698874	12.161943	0.080906	0.757998	12.161943
0.9	164	914	0.473639	0.772459	1.042869	0.071209	0.767262	18.168478	0.071209	0.771939	18.168478
perPairInter	countInterTC	countNoInterTC	minInterTC	avgInterTC	maxInterTC	minNoInterTC	avgNoInterTC	maxNoInterTC	minTotalTC	avgTotalTC	maxTotalTC
0.1	172	834	0.452918	0.753203	1.013232	0.060946	0.739067	12.805128	0.060946	0.740480	12.805128
0.2	176	832	0.457696	0.748322	1.018008	0.069421	0.682479	11.879581	0.069421	0.695647	11.879581
0.3	162	840	0.457023	0.748582	1.025043	0.073828	0.685503	9.681756	0.073828	0.704427	9.681756
0.4	154	802	0.453532	0.756155	1.009285	0.069254	0.727692	10.089474	0.069254	0.739077	10.089474
0.5	136	744	0.465391	0.745292	1.011189	0.092450	0.706725	12.467721	0.092450	0.726008	12.467721
0.6	154	820	0.454831	0.745605	1.014145	0.067833	0.652934	6.212298	0.067833	0.708536	6.212298
0.7	138	822	0.457112	0.753067	1.012875	0.059744	0.739597	11.781414	0.059744	0.749026	11.781414
0.8	180	870	0.453998	0.749818	1.081244	0.067798	0.721236	11.791560	0.067798	0.744102	11.791560
0.9	142	854	0.451885	0.752673	1.015569	0.081109	0.672038	18.168643	0.081109	0.744610	18.168643
perPairInter	countInterTT	countNoInterTT	minInterTT	avgInterTT	maxInterTT	minNoInterTT	avgNoInterTT	maxNoInterTT	minTotalTT	avgTotalTT	maxTotalTT
0.1	98	916	0.367096	0.435013	0.478811	0.161542	0.961766	13.943307	0.161542	0.909080	13.943307
0.2	124	892	0.336772	0.425974	0.484336	0.064063	0.886762	10.626794	0.064063	0.794650	10.626794
0.3	114	890	0.337496	0.424247	0.479187	0.062131	0.881683	12.704000	0.062131	0.744552	12.704000
0.4	94	938	0.372910	0.431235	0.479571	0.060422	0.858122	12.070423	0.060422	0.687367	12.070423
0.5	126	936	0.336753	0.427856	0.481165	0.081638	0.886573	14.341346	0.081638	0.657215	14.341346
0.6	134	904	0.341038	0.433330	0.477776	0.057055	0.963492	10.963492	0.057055	0.630365	10.963492
0.7	102	820	0.332448	0.432740	0.485270	0.065735	0.889582	9.626984	0.065735	0.626984	9.626984
0.8	88	888	0.335461	0.436193	0.479734	0.075558	0.795563	12.620192	0.075558	0.544267	12.620192
0.9	104	888	0.368328	0.433789	0.482883	0.072064	0.929475	14.850629	0.072064	0.483357	14.850629



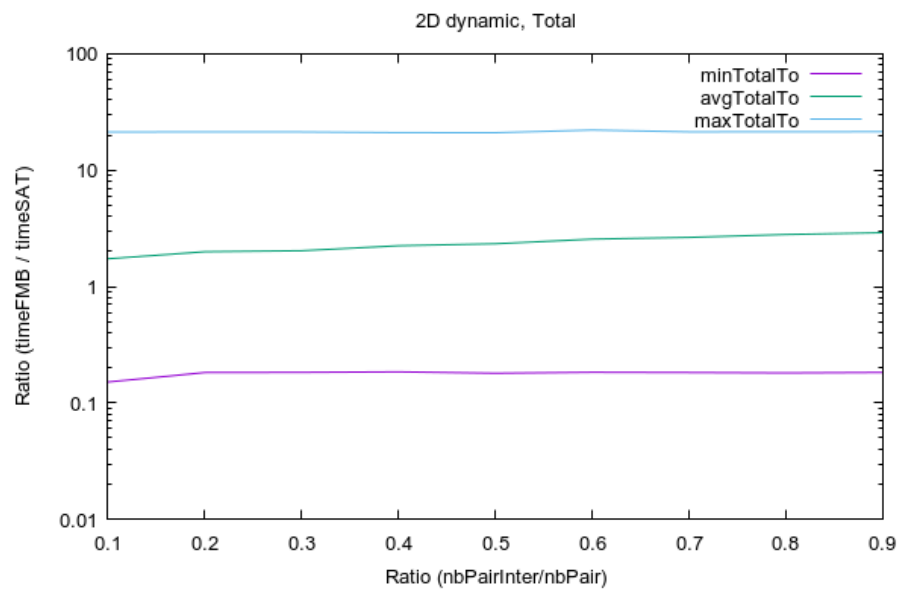


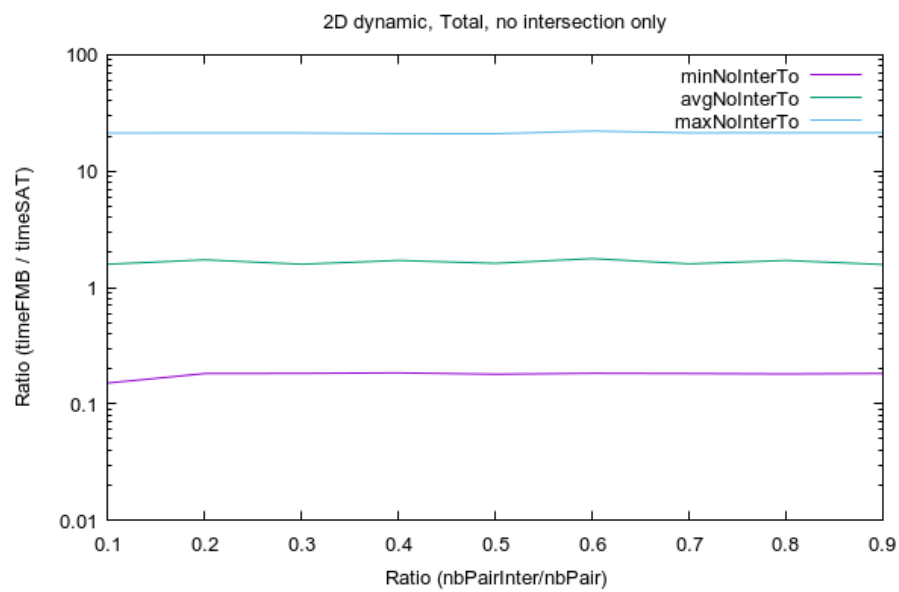
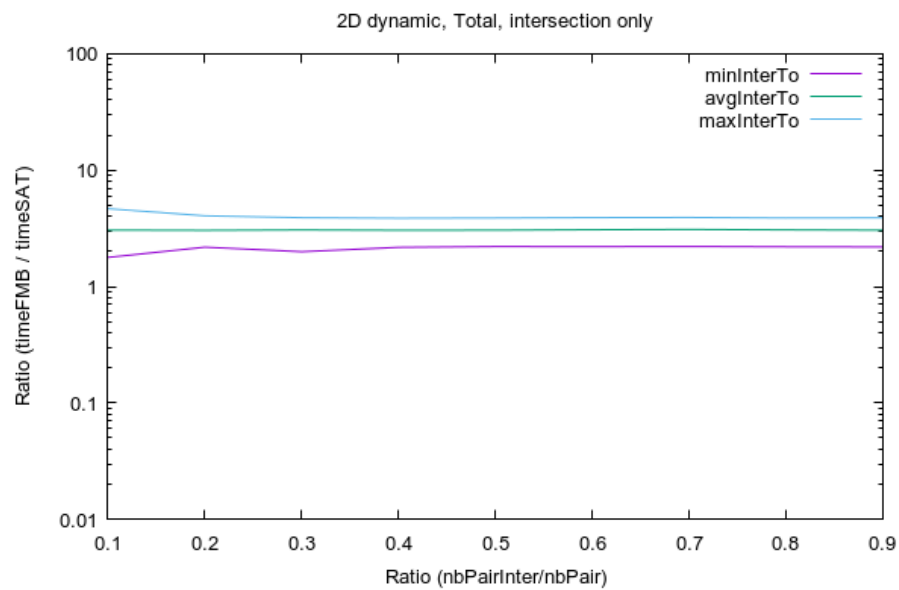


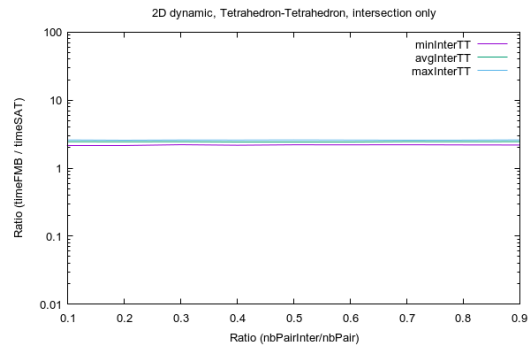
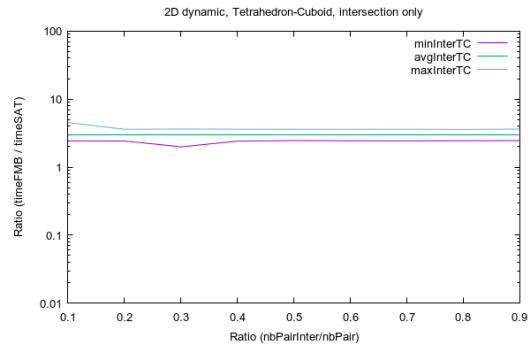
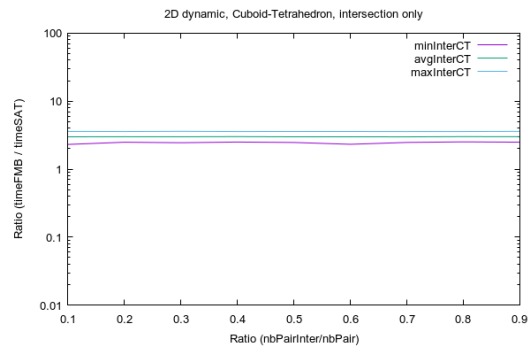
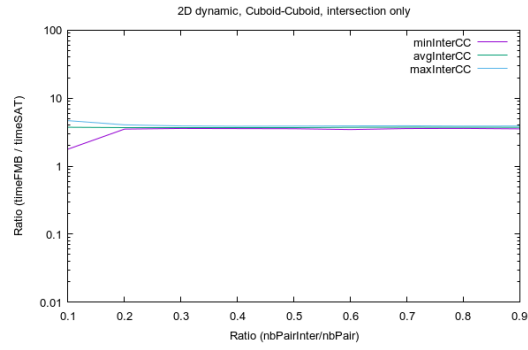


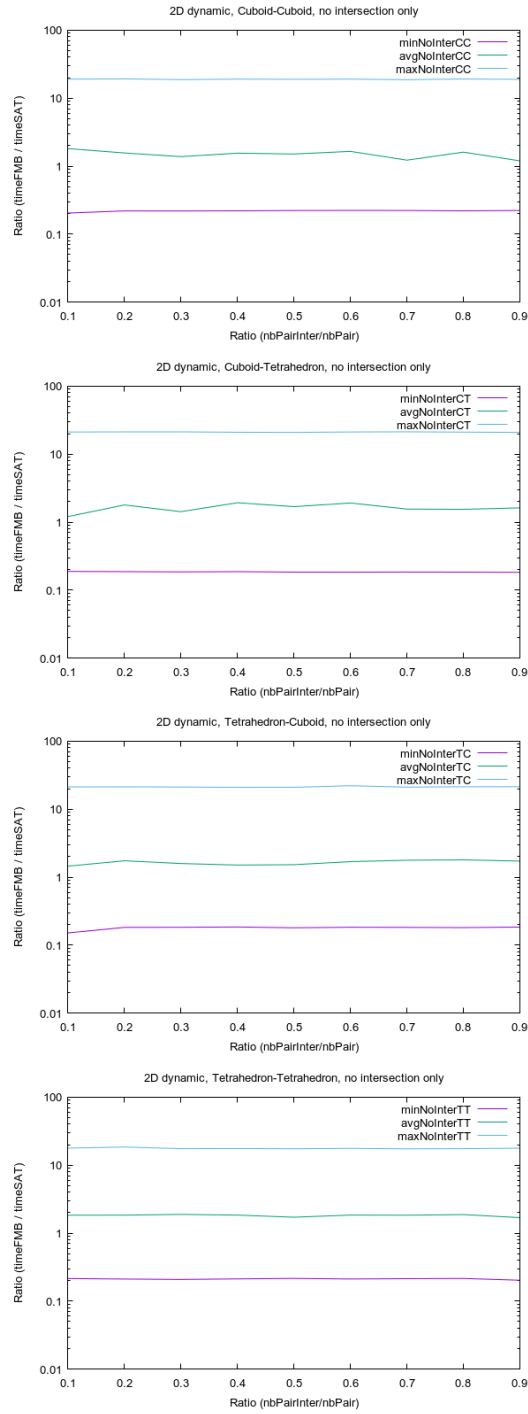
8.2.3 2D dynamic

perPairInter	countInterTo	countNoInterTo	minInterTo	avgInterTo	maxInterTo	minNoInterTo	avgNoInterTo	maxNoInterTo	minTotalTo	avgTotalTo	maxTotalTo
0.1	1508	2492	1.780150	3.056258	4.668988	0.151304	1.587200	21.269679	0.151304	1.734006	21.269679
0.2	1430	2570	2.180830	3.038811	4.051782	0.182547	1.731194	21.335286	0.182547	1.997346	21.335286
0.3	1492	2508	1.993233	3.902535	3.902535	0.183117	1.587633	21.304539	0.183117	2.030251	21.304539
0.4	1420	2580	2.174635	3.045617	3.863448	0.184885	1.709351	21.013043	0.184885	2.243858	21.013043
0.5	1468	2532	2.205622	3.049331	3.879654	0.180177	1.614648	20.992625	0.180177	2.319899	20.992625
0.6	1478	2522	2.203458	3.078241	3.907932	0.183318	1.773441	22.103293	0.183318	2.586321	22.103293
0.7	1520	2480	2.209376	3.094055	3.920449	0.182423	1.597889	21.338710	0.182423	2.645145	21.338710
0.8	1530	2470	2.195160	3.069844	3.880984	0.181217	1.710302	21.356204	0.181217	2.797935	21.356204
0.9	1462	2538	2.190191	3.048336	3.894050	0.182671	1.576506	21.407953	0.182671	2.901144	21.407953
perPairInter	countInterCC	countNoInterCC	minInterCC	avgInterCC	maxInterCC	minNoInterCC	avgNoInterCC	maxNoInterCC	minTotalCC	avgTotalCC	maxTotalCC
0.1	376	640	1.780150	3.735051	4.668988	0.204325	1.810475	19.025543	0.204325	2.002933	19.025543
0.2	352	662	3.511416	3.737762	4.051782	0.219499	1.5872241	19.202305	0.219499	1.997346	19.202305
0.3	406	584	3.587424	3.738052	3.902535	0.219112	1.380021	18.750627	0.219112	2.087430	18.750627
0.4	336	542	3.571290	3.737705	3.863448	0.220612	1.553991	19.049555	0.220612	2.427477	19.049555
0.5	360	632	3.551509	3.737641	3.879654	0.222539	1.508938	18.956633	0.222539	2.623280	18.956633
0.6	412	618	3.459689	3.729182	3.907932	0.223265	1.649491	19.054967	0.223265	2.897305	19.054967
0.7	456	614	3.574463	3.737380	3.920449	0.222720	1.225185	18.689139	0.222720	2.983722	18.689139
0.8	426	624	3.588979	3.731500	3.880984	0.220393	1.609081	19.150895	0.220393	3.307016	19.150895
0.9	358	540	3.536235	3.730124	3.894050	0.222629	1.196255	18.889313	0.222629	3.476737	18.889313
perPairInter	countInterCT	countNoInterCT	minInterCT	avgInterCT	maxInterCT	minNoInterCT	avgNoInterCT	maxNoInterCT	minTotalCT	avgTotalCT	maxTotalCT
0.1	402	566	2.311180	2.992712	3.590574	0.188566	1.209282	21.180636	0.188566	1.387625	21.180636
0.2	360	594	2.491339	3.001752	3.573454	0.187300	1.271137	21.271137	0.187300	2.033496	21.271137
0.3	356	576	2.446757	2.993513	3.603116	0.185436	1.425906	21.304539	0.185436	1.896188	21.304539
0.4	364	664	2.504857	3.007744	3.576159	0.187300	1.931313	21.013043	0.187300	2.361885	21.013043
0.5	374	650	2.469748	2.997187	3.590853	0.184187	1.691868	20.926829	0.184187	2.344528	20.926829
0.6	396	632	2.323215	2.995549	3.592653	0.183857	1.912404	21.200855	0.183857	2.562291	21.200855
0.7	348	636	2.471686	2.986385	3.568237	0.184817	1.557615	21.338710	0.184817	2.557754	21.338710
0.8	372	594	2.522422	3.006941	3.572020	0.184187	1.544286	21.092754	0.184187	2.714330	21.092754
0.9	392	676	2.495487	2.997798	3.594621	0.182671	1.622262	20.918338	0.182671	2.860244	20.918338
perPairInter	countInterTC	countNoInterTC	minInterTC	avgInterTC	maxInterTC	minNoInterTC	avgNoInterTC	maxNoInterTC	minTotalTC	avgTotalTC	maxTotalTC
0.1	398	644	2.433896	2.979596	4.518961	0.151304	1.451214	21.269679	0.151304	1.604052	21.269679
0.2	360	676	2.425671	2.985941	3.607592	0.182547	1.740955	21.335286	0.182547	1.989952	21.335286
0.3	368	628	1.993233	2.987021	3.632851	0.183117	1.585717	21.154685	0.183117	2.006108	21.154685
0.4	394	740	2.413815	2.987035	3.619544	0.184885	1.508355	20.943885	0.184885	2.099827	20.943885
0.5	392	582	2.465394	2.993655	3.609080	0.180177	1.526790	20.992625	0.180177	2.260227	20.992625
0.6	350	668	2.440424	2.982837	3.602825	0.183318	1.691449	22.103293	0.183318	2.466282	22.103293
0.7	382	600	2.436802	2.983518	3.614130	0.182423	1.773553	21.114493	0.182423	2.620528	21.114493
0.8	364	594	2.443041	2.989047	3.588730	0.181217	1.797271	21.356204	0.181217	2.750692	21.356204
0.9	372	682	2.459694	2.989725	3.625872	0.184450	1.722705	21.407953	0.184450	2.863023	21.407953
perPairInter	countInterTT	countNoInterTT	minInterTT	avgInterTT	maxInterTT	minNoInterTT	avgNoInterTT	maxNoInterTT	minTotalTT	avgTotalTT	maxTotalTT
0.1	332	642	2.180986	2.451809	2.599092	0.214847	1.834211	17.787728	0.214847	1.896971	17.787728
0.2	358	638	2.180830	2.442006	2.574832	0.213140	1.804075	18.504687	0.213140	1.960461	18.504687
0.3	362	720	2.211252	2.441586	2.539922	0.208524	1.887081	17.487013	0.208524	2.056433	17.487013
0.4	326	634	2.174635	2.445390	2.584892	0.212357	1.844303	17.572894	0.212357	2.084737	17.572894
0.5	342	668	2.205622	2.445619	2.596805	0.216017	1.716070	17.415858	0.216017	2.080845	17.415858
0.6	320	604	2.203458	2.446834	2.596805	0.211777	1.845539	17.662273	0.211777	2.206316	17.662273
0.7	334	630	2.209376	2.447709	2.577709	0.214062	1.833700	17.365260	0.214062	2.268153	17.365260
0.8	368	688	2.195160	2.447511	2.582475	0.215604	1.837446	17.498371	0.215604	2.333298	17.498371
0.9	340	640	2.190191	2.452804	2.599661	0.202863	1.633221	17.777961	0.202863	2.376846	17.777961



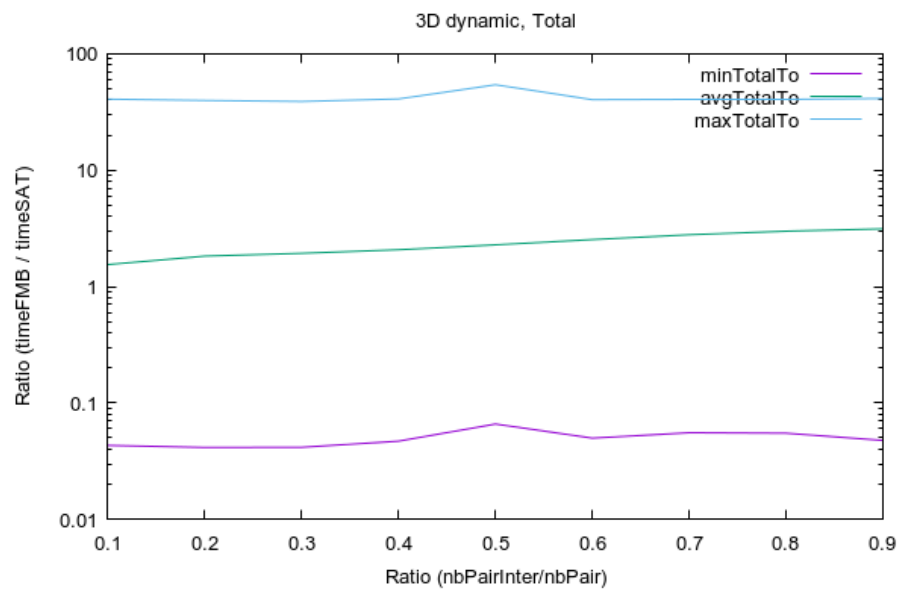


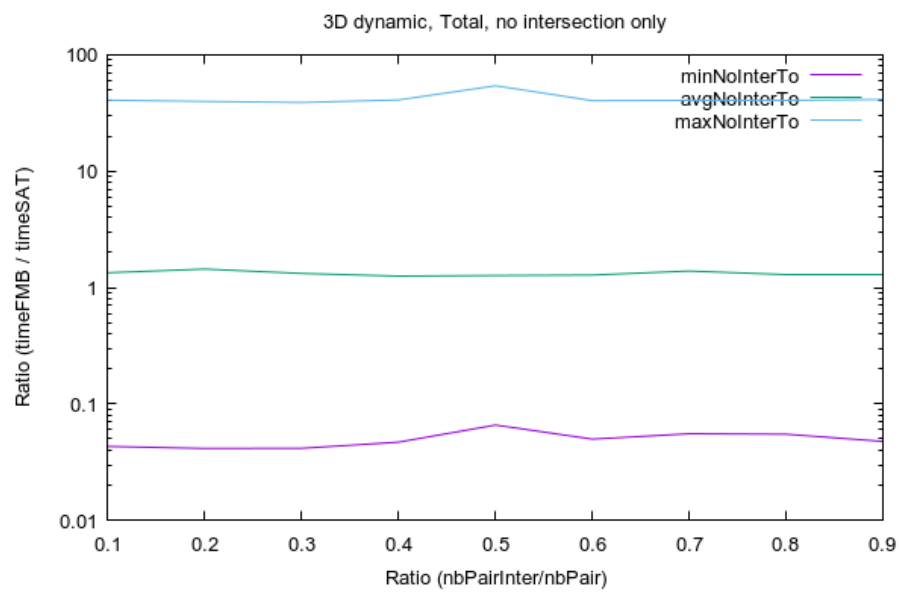
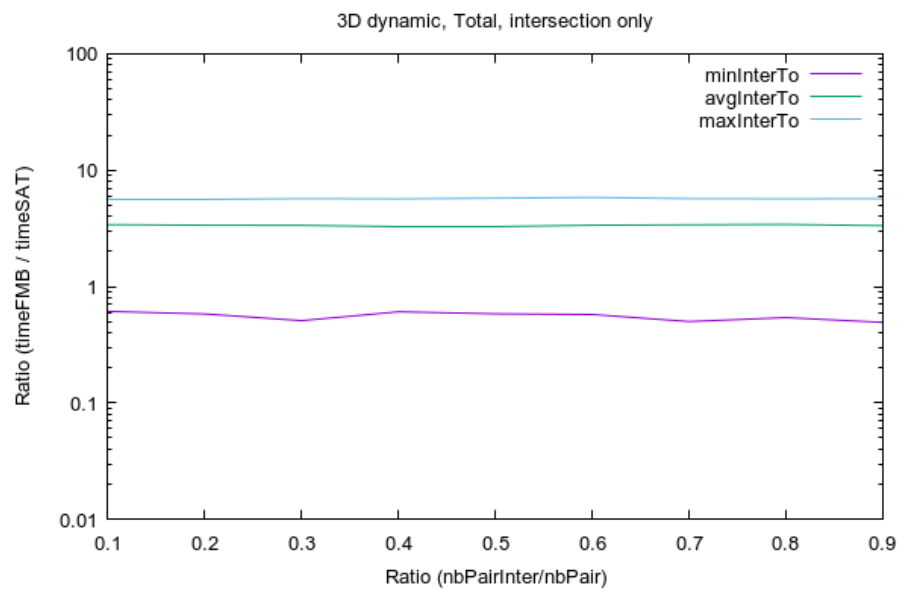


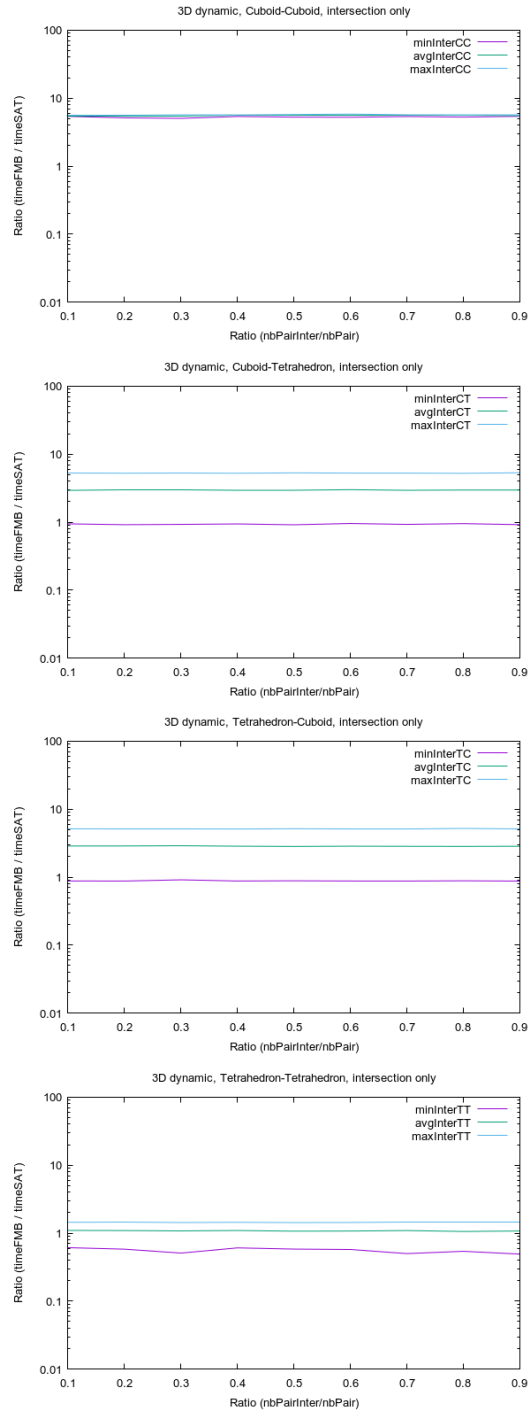


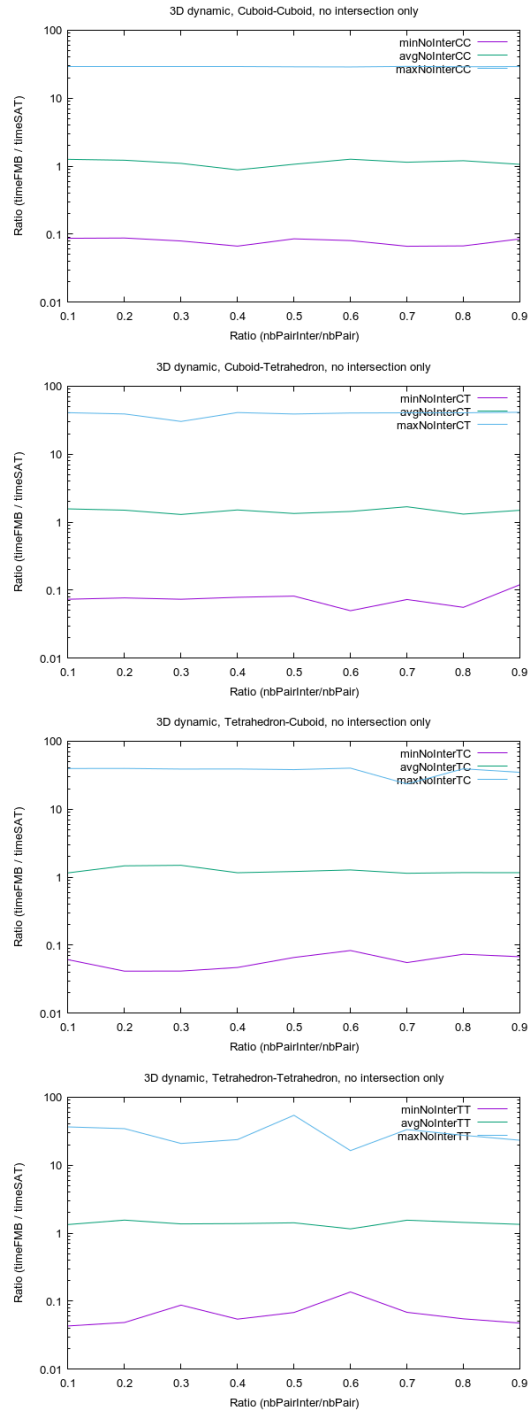
8.2.4 3D dynamic

perPairInter	countInterTo	countNoInterTo	minInterTo	avgInterTo	maxInterTo	minNoInterTo	avgNoInterTo	maxNoInterTo	minTotalTo	avgTotalTo	maxTotalTo
0.1	1050	2950	0.611546	3.403477	5.682570	0.043169	1.339379	40.724359	0.043169	1.545788	40.724359
0.2	1150	2860	0.581245	5.682302	5.682302	0.041481	1.441354	39.717903	0.041481	1.826300	39.717903
0.3	1044	2956	0.509138	3.357135	5.694552	0.041609	1.321157	38.902219	0.041609	1.931590	38.902219
0.4	1010	2990	0.607386	3.294303	5.671153	0.046938	1.253858	40.949830	0.046938	2.070066	40.949830
0.5	964	3036	0.583427	5.748242	5.748242	0.065865	1.268219	54.170972	0.065865	2.283108	54.170972
0.6	934	3066	0.575035	3.368895	5.836756	0.049824	1.281187	40.319650	0.049824	2.538612	40.319650
0.7	1052	2948	0.500555	3.369659	5.697016	0.055427	1.386098	40.987296	0.055427	2.788590	40.987296
0.8	1008	2992	0.541019	3.415734	5.682426	0.054948	1.282366	40.992742	0.054948	2.989060	40.992742
0.9	1132	2868	0.492177	3.338711	5.692532	0.047671	1.282601	41.295771	0.047671	3.133100	41.295771
perPairInter	countInterCC	countNoInterCC	minInterCC	avgInterCC	maxInterCC	minNoInterCC	avgNoInterCC	maxNoInterCC	minTotalCC	avgTotalCC	maxTotalCC
0.1	336	722	5.400994	5.531338	5.682570	0.086898	1.259771	29.285743	0.086898	1.686928	29.285743
0.2	366	670	5.134412	5.529890	5.682302	0.087649	1.222985	29.297567	0.087649	2.084366	29.297567
0.3	304	666	5.046581	5.532482	5.694552	0.079365	1.099077	28.807723	0.079365	2.429100	28.807723
0.4	284	636	5.376225	5.533067	5.671153	0.066632	0.813031	28.813031	0.066632	2.741985	28.813031
0.5	286	676	5.264725	5.536360	5.748242	0.085396	1.066285	28.906555	0.085396	3.301322	28.906555
0.6	282	722	5.236274	5.536933	5.836756	0.080387	1.264591	28.799882	0.080387	3.827996	28.799882
0.7	340	678	5.364676	5.532860	5.697016	0.065392	1.142460	29.327524	0.065392	4.215740	29.327524
0.8	314	696	5.270684	5.535163	5.682426	0.067150	1.201623	28.970309	0.067150	4.668455	28.970309
0.9	324	616	5.406529	5.531641	5.692532	0.084962	1.063572	29.128068	0.084962	5.094834	29.128068
perPairInter	countInterCT	countNoInterCT	minInterCT	avgInterCT	maxInterCT	minNoInterCT	avgNoInterCT	maxNoInterCT	minTotalCT	avgTotalCT	maxTotalCT
0.1	266	760	0.944378	2.932406	5.279508	0.073448	1.566604	40.724359	0.073448	1.703184	40.724359
0.2	280	696	0.918672	2.998924	5.245975	0.076898	1.501284	38.991334	0.076898	1.800812	38.991334
0.3	278	760	0.927542	2.998105	5.274782	0.073451	1.297223	30.310530	0.073451	1.807488	30.310530
0.4	250	796	0.941951	2.940583	5.238337	0.078453	1.512013	40.949830	0.078453	2.083441	40.949830
0.5	250	740	0.915604	2.945032	5.312305	0.081730	1.341645	38.939261	0.081730	2.143339	38.939261
0.6	242	748	0.955924	3.005952	5.272476	0.049824	1.437962	40.319650	0.049824	2.378756	40.319650
0.7	254	744	0.927799	2.941627	5.264182	0.072824	1.685404	40.987296	0.072824	2.564760	40.987296
0.8	272	756	0.952229	2.967806	5.225244	0.055889	1.312658	40.992742	0.055889	2.636776	40.992742
0.9	290	776	0.919399	2.964790	5.331673	0.120109	1.495120	41.295771	0.120109	2.817823	41.295771
perPairInter	countInterTC	countNoInterTC	minInterTC	avgInterTC	maxInterTC	minNoInterTC	avgNoInterTC	maxNoInterTC	minTotalTC	avgTotalTC	maxTotalTC
0.1	250	630	0.879498	2.871628	5.142451	0.061123	1.156047	39.633172	0.061123	1.327605	39.633172
0.2	256	742	0.876829	2.876605	5.123317	0.041481	1.468340	39.717903	0.041481	1.749993	39.717903
0.3	270	744	0.914219	2.887248	5.123935	0.041609	1.494517	38.902219	0.041609	1.915336	38.902219
0.4	284	764	0.877512	2.853995	5.108823	0.046938	1.161255	38.991530	0.046938	1.838351	38.991530
0.5	228	782	0.884356	2.831219	5.157719	0.065865	1.210798	38.123761	0.065865	2.021008	38.123761
0.6	234	774	0.877924	2.856921	5.125343	0.083409	1.277160	40.132104	0.083409	2.225017	40.132104
0.7	250	754	0.877076	2.841177	5.110771	0.055427	1.140408	23.507028	0.055427	2.330947	23.507028
0.8	254	764	0.884354	2.834126	5.204659	0.073463	1.166164	39.252868	0.073463	2.500533	39.252868
0.9	322	684	0.875361	2.845824	5.130184	0.067605	1.153353	34.798109	0.067605	2.677577	34.798109
perPairInter	countInterTT	countNoInterTT	minInterTT	avgInterTT	maxInterTT	minNoInterTT	avgNoInterTT	maxNoInterTT	minTotalTT	avgTotalTT	maxTotalTT
0.1	198	838	0.611546	1.096939	1.442664	0.043169	1.339718	36.505896	0.043169	1.315440	36.505896
0.2	248	742	0.581245	1.092542	1.455034	0.048549	1.555332	34.467133	0.048549	1.422774	34.467133
0.3	192	786	0.509138	1.079400	1.432991	0.087488	1.386375	20.867700	0.087488	1.281692	20.867700
0.4	192	794	0.607386	1.094660	1.442829	0.054383	1.382609	23.751445	0.054383	1.267429	23.751445
0.5	200	838	0.583427	1.070472	1.430513	0.068133	1.417092	54.170972	0.068133	1.245166	54.170972
0.6	176	822	0.575035	1.074947	1.435507	0.136566	1.156987	16.393152	0.136566	1.107667	16.393152
0.7	208	772	0.500555	1.092696	1.455644	0.084557	1.230361	33.379976	0.084557	1.337976	33.379976
0.8	168	776	0.541019	1.058972	1.441002	0.054948	1.551581	27.483803	0.054948	1.135114	27.483803
0.9	196	792	0.492177	1.076659	1.442433	0.047671	1.347718	23.274742	0.047671	1.103765	23.274742









9 Conclusion

The validation proves that the FMB algorithm correctly identifies intersection of pairs of Frames in accordance with the results of the SAT algorithm.

The qualification shows that the FMB is 1.2 to 1.8 times slower than the SAT algorithm in the 2D dynamic case. However it is around 2 times faster in the 3D static case, and up to 1.25 times faster in 3D dynamic and up to 1.1 times faster in the 2D static case if the percentage of tested pairs in intersection is less than, respectively, around 40% and 25%.

On one given pair of Frame, the relative speed of the FMB algorithm varies widely, from around 20 times slower to 50 times faster. This is explained by the way the 2 algorithms works: they both make the assumption that the Frames are intersecting and run through a series of tests to try to prove it wrong. This leads to best cases and worst cases for both algorithm: a non intersecting detected right from the first test, or one detected by the last test. These best and worst cases are different for the two algorithms as the tests they performed are completely different. But in average, the FMB algorithm has the advantage for all but the 2D dynamic case.

10 Annex

10.1 Runtime environment

Results introduce in this paper have been produced by compiling and running the corresponding algorithms in the following environment:

```
uname -v

#40~18.04.1-Ubuntu SMP Thu Nov 14 12:06:39 UTC 2019

=====

lshw -short
```

H/W path	Device	Class	Description
		system	VC65-C1
/0		bus	VC65-C1
/0/0		memory	64KiB BIOS
/0/2f		memory	16GiB System Memory
/0/2f/0		memory	[empty]
/0/2f/1		memory	16GiB SODIMM DDR4 Synchronous 2400
			MHz (0.4 ns)

/0/39		memory	384KiB L1 cache
/0/3a		memory	1536KiB L2 cache
/0/3b		memory	12MiB L3 cache
/0/3c		processor	Intel(R) Core(TM) i7-8700T CPU @
2.40GHz			
/0/100		bridge	8th Gen Core Processor Host Bridge
/DRAM Registers			
/0/100/2		display	Intel Corporation
/0/100/12		generic	Cannon Lake PCH Thermal Controller
/0/100/14		bus	Cannon Lake PCH USB 3.1 xHCI Host
Controller			
/0/100/14/0	usb1	bus	xHCI Host Controller
/0/100/14/0/5		input	ELECOM Wired Keyboard
/0/100/14/0/6		input	PTZ-630
/0/100/14/0/7		generic	USB2.0-CRW
/0/100/14/0/e		communication	Bluetooth wireless interface
/0/100/14/1	usb2	bus	xHCI Host Controller
/0/100/14.2		memory	RAM memory
/0/100/14.3	wlo1	network	Wireless-AC 9560 [Jefferson Peak]
/0/100/16		communication	Cannon Lake PCH HECI Controller
/0/100/17		storage	Cannon Lake PCH SATA AHCI
Controller			
/0/100/1f		bridge	Intel Corporation
/0/100/1f.3		multimedia	Cannon Lake PCH cAVS
/0/100/1f.4		bus	Cannon Lake PCH SMBus Controller
/0/100/1f.5		bus	Cannon Lake PCH SPI Controller
/0/100/1f.6	eno2	network	Ethernet Connection (7) I219-V
/0/1	scsi0	storage	
/0/1/0.0.0	/dev/sda	disk	128GB HFS128G39TND-N21
/0/1/0.0.0/1		volume	99MiB Windows FAT volume
/0/1/0.0.0/2	/dev/sda2	volume	15MiB reserved partition
/0/1/0.0.0/3	/dev/sda3	volume	83GiB Windows NTFS volume
/0/1/0.0.0/4	/dev/sda4	volume	499MiB Windows NTFS volume
/0/1/0.0.0/5	/dev/sda5	volume	35GiB EXT4 volume
/0/2	scsi2	storage	
/0/2/0.0.0	/dev/sdb	disk	500GB ST500LM034-2GH17
/0/2/0.0.0/1	/dev/sdb1	volume	463GiB EXT4 volume
/0/2/0.0.0/2	/dev/sdb2	volume	499MiB Windows FAT volume
/0/3	scsi5	storage	
/0/3/0.0.0	/dev/cdrom	disk	BD-RE BU50N
/1		power	To Be Filled By O.E.M.

=====

lscpu

```

Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                 12
On-line CPU(s) list:   0-11
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):              1
NUMA node(s):          1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  158
Model name:             Intel(R) Core(TM) i7-8700T CPU @ 2.40GHz
Stepping:               10
CPU MHz:                2216.548
CPU max MHz:            4000.0000

```

```

CPU min MHz:      800.0000
BogoMIPS:         4800.00
Virtualization:   VT-x
L1d cache:       32K
L1i cache:       32K
L2 cache:        256K
L3 cache:        12288K
NUMA node0 CPU(s): 0-11
Flags:            fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
                  syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
                  rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni
                  pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr
                  pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes
                  xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb
                  invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept
                  vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid
                  rtm mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1
                  xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp
                  md_clear flush_l1d

```

=====

```
gcc -v
```

```
Using built-in specs.
```

```
COLLECT_GCC=gcc
```

```
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
```

```
OFFLOAD_TARGET_NAMES=nvptx-none
```

```
OFFLOAD_TARGET_DEFAULT=1
```

```
Target: x86_64-linux-gnu
```

```

Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.4.0-1
ubuntu18.04.1' --with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs
--enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++ --prefix=/
usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=
x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir
=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/
usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-
libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi
=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx
--enable-plugin --enable-default-pie --with-system-zlib --with-target-
system-zlib --enable-objc-gc=auto --enable-multiarch --disable-werror --
with-arch=32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --
enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none
--without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu
--host=x86_64-linux-gnu --target=x86_64-linux-gnu

```

```
Thread model: posix
```

```
gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)
```

10.2 SAT implementation

In this section I introduce the code of the implementation of the SAT algorithm, used to validate and qualify the FMB algorithm.

10.2.1 Header

```

#ifndef __SAT_H_
#define __SAT_H_

#include <stdbool.h>
#include <string.h>
#include "frame.h"

// ----- Functions declaration -----

// Test for intersection between 2D Frame that and 2D Frame tho
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2D(
    const Frame2D* const that,
    const Frame2D* const tho);

// Test for intersection between moving 2D Frame that and 2D
// Frame tho
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2DTime(
    const Frame2DTime* const that,
    const Frame2DTime* const tho);

// Test for intersection between 3D Frame that and 3D Frame tho
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3D(
    const Frame3D* const that,
    const Frame3D* const tho);

// Test for intersection between moving 3D Frame that and 3D
// Frame tho
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3DTime(
    const Frame3DTime* const that,
    const Frame3DTime* const tho);

#endif

```

10.2.2 Body

```

#include "sat.h"

// ----- Macros -----

#define EPSILON 0.0000001

// ----- Functions declaration -----

// Check the intersection constraint along one axis for 3D Frames
bool CheckAxis3D(
    const Frame3D* const that,
    const Frame3D* const tho,
    const double* const axis);

// Check the intersection constraint along one axis for moving 3D Frames
bool CheckAxis3DTime(
    const Frame3DTime* const that,
    const Frame3DTime* const tho,
    const double* const axis,
    const double* const relSpeed);

// ----- Functions implementation -----

```

```

// Test for intersection between 2D Frame that and 2D Frame tho
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2D(
    const Frame2D* const that,
    const Frame2D* const tho) {

    // Declare a variable to loop on Frames and commonalize code
    const Frame2D* frameEdge = that;

    // Loop to commonalize code when checking SAT based on that's edges
    // and then tho's edges
    for (int iFrame = 2;
        iFrame--;) {

        // Shortcuts
        FrameType frameEdgeType = frameEdge->type;
        const double* frameEdgeCompA = frameEdge->comp[0];
        const double* frameEdgeCompB = frameEdge->comp[1];

        // Declare a variable to memorize the number of edges, by default 2
        int nbEdges = 2;

        // Declare a variable to memorize the third edge in case of
        // tetrahedron
        double thirdEdge[2];

        // If the frame is a tetrahedron
        if (frameEdgeType == FrameTetrahedron) {

            // Initialise the third edge
            thirdEdge[0] = frameEdgeCompB[0] - frameEdgeCompA[0];
            thirdEdge[1] = frameEdgeCompB[1] - frameEdgeCompA[1];

            // Correct the number of edges
            nbEdges = 3;

        }

        // Loop on the frame's edges
        for (int iEdge = nbEdges;
            iEdge--;) {

            // Get the current edge
            const double* edge =
                (iEdge == 2 ? thirdEdge : frameEdge->comp[iEdge]);

            // Declare variables to memorize the boundaries of projection
            // of the two frames on the current edge
            double bdgBoxA[2];
            double bdgBoxB[2];

            // Declare two variables to loop on Frames and commonalize code
            const Frame2D* frame = that;
            double* bdgBox = bdgBoxA;

            // Loop on Frames
            for (int iFrame = 2;
                iFrame--;) {

                // Shortcuts
                const double* frameOrig = frame->orig;

```

```

const double* frameCompA = frame->comp[0];
const double* frameCompB = frame->comp[1];
FrameType frameType = frame->type;

// Get the number of vertices of frame
int nbVertices = (frameType == FrameTetrahedron ? 3 : 4);

// Declare a variable to memorize if the current vertex is
// the first in the loop, used to initialize the boundaries
bool firstVertex = true;

// Loop on vertices of the frame
for (int iVertex = nbVertices;
     iVertex--;) {

    // Get the vertex
    double vertex[2];
    vertex[0] = frameOrig[0];
    vertex[1] = frameOrig[1];
    switch (iVertex) {
        case 3:
            vertex[0] += frameCompA[0] + frameCompB[0];
            vertex[1] += frameCompA[1] + frameCompB[1];
            break;
        case 2:
            vertex[0] += frameCompA[0];
            vertex[1] += frameCompA[1];
            break;
        case 1:
            vertex[0] += frameCompB[0];
            vertex[1] += frameCompB[1];
            break;
        default:
            break;
    }

    // Get the projection of the vertex on the normal of the edge
    // Orientation of the normal doesn't matter, so we
    // use arbitrarily the normal (edge[1], -edge[0])
    double proj = vertex[0] * edge[1] - vertex[1] * edge[0];

    // If it's the first vertex
    if (firstVertex == true) {

        // Initialize the boundaries of the projection of the
        // Frame on the edge
        bdgBox[0] = proj;
        bdgBox[1] = proj;

        // Update the flag to memorize we did the first vertex
        firstVertex = false;
    }

    // Else, it's not the first vertex
    else {

        // Update the boundaries of the projection of the Frame on
        // the edge
        if (bdgBox[0] > proj)
            bdgBox[0] = proj;

        if (bdgBox[1] < proj)
            bdgBox[1] = proj;
    }
}

```

```

    }

}

// Switch the frame to check the vertices of the second Frame
frame = tho;
bdgBox = bdgBoxB;

}

// If the projections of the two frames on the edge are
// not intersecting
if (bdgBoxB[1] < bdgBoxA[0] ||
    bdgBoxA[1] < bdgBoxB[0]) {

    // There exists an axis which separates the Frames,
    // thus they are not in intersection
    return false;

}

}

// Switch the frames to test against the second Frame's edges
frameEdge = tho;

}

// If we reaches here, it means the two Frames are intersecting
return true;

}

// Test for intersection between moving 2D Frame that and 2D
// Frame tho
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2DTime(
    const Frame2DTime* const that,
    const Frame2DTime* const tho) {

    // Declare a variable to loop on Frames and commonalize code
    const Frame2DTime* frameEdge = that;

    // Declare a variable to memorize the speed of tho relative to that
    double relSpeed[2];
    relSpeed[0] = tho->speed[0] - that->speed[0];
    relSpeed[1] = tho->speed[1] - that->speed[1];

    // Loop to commonalize code when checking SAT based on that's edges
    // and then tho's edges
    for (int iFrame = 2;
        iFrame--;) {

        // Shortcuts
        FrameType frameEdgeType = frameEdge->type;
        const double* frameEdgeCompA = frameEdge->comp[0];
        const double* frameEdgeCompB = frameEdge->comp[1];

        // Declare a variable to memorize the number of edges, by default 2
        int nbEdges = 2;

```

```

// Declare a variable to memorize the third edge in case of
// tetrahedron
double thirdEdge[2];

// If the frame is a tetrahedron
if (frameEdgeType == FrameTetrahedron) {

    // Initialise the third edge
    thirdEdge[0] = frameEdgeCompB[0] - frameEdgeCompA[0];
    thirdEdge[1] = frameEdgeCompB[1] - frameEdgeCompA[1];

    // Correct the number of edges
    nbEdges = 3;

}

// If the current frame is the second frame
if (iFrame == 1) {

    // Add one more edge to take into account the movement
    // of the relative to that
    ++nbEdges;

}

// Loop on the frame's edges
for (int iEdge = nbEdges;
     iEdge--;) {

    // Get the current edge
    const double* edge =
        (iEdge == 3 ? relSpeed :
         (iEdge == 2 ?
          (frameEdgeType == FrameTetrahedron ? thirdEdge : relSpeed) :
          frameEdge->comp[iEdge]));

    // Declare variables to memorize the boundaries of projection
    // of the two frames on the current edge
    double bdgBoxA[2];
    double bdgBoxB[2];

    // Declare two variables to loop on Frames and commonalize code
    const Frame2DTime* frame = that;
    double* bdgBox = bdgBoxA;

    // Loop on Frames
    for (int iFrame = 2;
         iFrame--;) {

        // Shortcuts
        const double* frameOrig = frame->orig;
        const double* frameCompA = frame->comp[0];
        const double* frameCompB = frame->comp[1];
        FrameType frameType = frame->type;

        // Get the number of vertices of frame
        int nbVertices = (frameType == FrameTetrahedron ? 3 : 4);

        // Declare a variable to memorize if the current vertex is
        // the first in the loop, used to initialize the boundaries
        bool firstVertex = true;

```



```

// Loop on vertices of the frame
for (int iVertex = nbVertices;
     iVertex--;) {

    // Get the vertex
    double vertex[2];
    vertex[0] = frameOrig[0];
    vertex[1] = frameOrig[1];
    switch (iVertex) {
        case 3:
            vertex[0] += frameCompA[0] + frameCompB[0];
            vertex[1] += frameCompA[1] + frameCompB[1];
            break;
        case 2:
            vertex[0] += frameCompA[0];
            vertex[1] += frameCompA[1];
            break;
        case 1:
            vertex[0] += frameCompB[0];
            vertex[1] += frameCompB[1];
            break;
        default:
            break;
    }

    // Get the projection of the vertex on the normal of the edge
    // Orientation of the normal doesn't matter, so we
    // use arbitrarily the normal (edge[1], -edge[0])
    double proj = vertex[0] * edge[1] - vertex[1] * edge[0];

    // If it's the first vertex
    if (firstVertex == true) {

        // Initialize the boundaries of the projection of the
        // Frame on the edge
        bdgBox[0] = proj;
        bdgBox[1] = proj;

        // Update the flag to memorize we did the first vertex
        firstVertex = false;
    }

    // Else, it's not the first vertex
    } else {

        // Update the boundaries of the projection of the Frame on
        // the edge
        if (bdgBox[0] > proj)
            bdgBox[0] = proj;

        if (bdgBox[1] < proj)
            bdgBox[1] = proj;
    }

    // If we are checking the second frame's vertices
    if (frame == tho) {

        // Check also the vertices moved by the relative speed
        vertex[0] += relSpeed[0];
        vertex[1] += relSpeed[1];

        proj = vertex[0] * edge[1] - vertex[1] * edge[0];
    }
}

```

```

        if (bdgBox[0] > proj)
            bdgBox[0] = proj;

        if (bdgBox[1] < proj)
            bdgBox[1] = proj;
    }

}

// Switch the frame to check the vertices of the second Frame
frame = tho;
bdgBox = bdgBoxB;

}

// If the projections of the two frames on the edge are
// not intersecting
if (bdgBoxB[1] < bdgBoxA[0] ||
    bdgBoxA[1] < bdgBoxB[0]) {

    // There exists an axis which separates the Frames,
    // thus they are not in intersection
    return false;

}

}

// Switch the frames to test against the second Frame's edges
frameEdge = tho;

}

// If we reaches here, it means the two Frames are intersecting
return true;

}

// Test for intersection between 3D Frame that and 3D Frame tho
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3D(
    const Frame3D* const that,
    const Frame3D* const tho) {

    // Declare two variables to memorize the opposite edges in case
    // of tetrahedron
    double oppEdgesThat[3][3];
    double oppEdgesTho[3][3];

    // Declare two variables to memorize the number of edges, by default 3
    int nbEdgesThat = 3;
    int nbEdgesTho = 3;

    // If the first Frame is a tetrahedron
    if (that->type == FrameTetrahedron) {

        // Shortcuts
        const double* frameCompA = that->comp[0];
        const double* frameCompB = that->comp[1];
        const double* frameCompC = that->comp[2];
    }
}

```

```

// Initialise the opposite edges
oppEdgesThat[0][0] = frameCompB[0] - frameCompA[0];
oppEdgesThat[0][1] = frameCompB[1] - frameCompA[1];
oppEdgesThat[0][2] = frameCompB[2] - frameCompA[2];

oppEdgesThat[1][0] = frameCompB[0] - frameCompC[0];
oppEdgesThat[1][1] = frameCompB[1] - frameCompC[1];
oppEdgesThat[1][2] = frameCompB[2] - frameCompC[2];

oppEdgesThat[2][0] = frameCompC[0] - frameCompA[0];
oppEdgesThat[2][1] = frameCompC[1] - frameCompA[1];
oppEdgesThat[2][2] = frameCompC[2] - frameCompA[2];

// Correct the number of edges
nbEdgesThat = 6;
}

// If the second Frame is a tetrahedron
if (tho->type == FrameTetrahedron) {

    // Shortcuts
    const double* frameCompA = tho->comp[0];
    const double* frameCompB = tho->comp[1];
    const double* frameCompC = tho->comp[2];

    // Initialise the opposite edges
    oppEdgesTho[0][0] = frameCompB[0] - frameCompA[0];
    oppEdgesTho[0][1] = frameCompB[1] - frameCompA[1];
    oppEdgesTho[0][2] = frameCompB[2] - frameCompA[2];

    oppEdgesTho[1][0] = frameCompB[0] - frameCompC[0];
    oppEdgesTho[1][1] = frameCompB[1] - frameCompC[1];
    oppEdgesTho[1][2] = frameCompB[2] - frameCompC[2];

    oppEdgesTho[2][0] = frameCompC[0] - frameCompA[0];
    oppEdgesTho[2][1] = frameCompC[1] - frameCompA[1];
    oppEdgesTho[2][2] = frameCompC[2] - frameCompA[2];

    // Correct the number of edges
    nbEdgesTho = 6;
}

// Declare variables to loop on Frames and commonalize code
const Frame3D* frame = that;
const double (*oppEdgesA)[3] = oppEdgesThat;

// Loop to commonalize code when checking SAT based on that's edges
// and then tho's edges
for (int iFrame = 2;
     iFrame--;) {

    // Shortcuts
    FrameType frameType = frame->type;
    const double* frameCompA = frame->comp[0];
    const double* frameCompB = frame->comp[1];
    const double* frameCompC = frame->comp[2];

    // Declare a variable to memorize the number of faces, by default 3
    int nbFaces = 3;

```

```

// Declare a variable to memorize the normal to faces
// Arrangement is normFaces[iFace][iAxis]
double normFaces[4][3];

// Initialise the normal to faces
normFaces[0][0] =
    frameCompA[1] * frameCompB[2] -
    frameCompA[2] * frameCompB[1];
normFaces[0][1] =
    frameCompA[2] * frameCompB[0] -
    frameCompA[0] * frameCompB[2];
normFaces[0][2] =
    frameCompA[0] * frameCompB[1] -
    frameCompA[1] * frameCompB[0];

normFaces[1][0] =
    frameCompA[1] * frameCompC[2] -
    frameCompA[2] * frameCompC[1];
normFaces[1][1] =
    frameCompA[2] * frameCompC[0] -
    frameCompA[0] * frameCompC[2];
normFaces[1][2] =
    frameCompA[0] * frameCompC[1] -
    frameCompA[1] * frameCompC[0];

normFaces[2][0] =
    frameCompC[1] * frameCompB[2] -
    frameCompC[2] * frameCompB[1];
normFaces[2][1] =
    frameCompC[2] * frameCompB[0] -
    frameCompC[0] * frameCompB[2];
normFaces[2][2] =
    frameCompC[0] * frameCompB[1] -
    frameCompC[1] * frameCompB[0];

// If the frame is a tetrahedron
if (frameType == FrameTetrahedron) {

    // Shortcuts
    const double* oppEdgeA = oppEdgesA[0];
    const double* oppEdgeB = oppEdgesA[1];

    // Initialise the normal to the opposite face
    normFaces[3][0] =
        oppEdgeA[1] * oppEdgeB[2] -
        oppEdgeA[2] * oppEdgeB[1];
    normFaces[3][1] =
        oppEdgeA[2] * oppEdgeB[0] -
        oppEdgeA[0] * oppEdgeB[2];
    normFaces[3][2] =
        oppEdgeA[0] * oppEdgeB[1] -
        oppEdgeA[1] * oppEdgeB[0];

    // Correct the number of faces
    nbFaces = 4;
}

// Loop on the frame's faces
for (int iFace = nbFaces;
    iFace--;) {

```

```

// Check against the current face's normal
bool isIntersection =
    CheckAxis3D(
        that,
        tho,
        normFaces[iFace]);

// If the axis is separating the Frames
if (isIntersection == false) {

    // The Frames are not in intersection,
    // terminate the test
    return false;

}

}

// Switch the frame to test against the second Frame
frame = tho;
oppEdgesA = oppEdgesTho;

}

// Loop on the pair of edges between the two frames
for (int iEdgeThat = nbEdgesThat;
     iEdgeThat--;) {

    // Get the first edge
    const double* edgeThat =
        (iEdgeThat < 3 ?
         that->comp[iEdgeThat] :
         oppEdgesThat[iEdgeThat - 3]);

    for (int iEdgeTho = nbEdgesTho;
         iEdgeTho--;) {

        // Get the second edge
        const double* edgeTho =
            (iEdgeTho < 3 ?
             tho->comp[iEdgeTho] :
             oppEdgesTho[iEdgeTho - 3]);

        // Get the cross product of the two edges
        double axis[3];
        axis[0] = edgeThat[1] * edgeTho[2] - edgeThat[2] * edgeTho[1];
        axis[1] = edgeThat[2] * edgeTho[0] - edgeThat[0] * edgeTho[2];
        axis[2] = edgeThat[0] * edgeTho[1] - edgeThat[1] * edgeTho[0];

        // Check against the cross product of the two edges
        bool isIntersection =
            CheckAxis3D(
                that,
                tho,
                axis);

        // If the axis is separating the Frames
        if (isIntersection == false) {

            // The Frames are not in intersection,
            // terminate the test

```

```

        return false;

    }

}

// If we reaches here, it means the two Frames are intersecting
return true;
}

// Test for intersection between moving 3D Frame that and 3D
// Frame tho
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3DTime(
    const Frame3DTime* const that,
    const Frame3DTime* const tho) {

    // Declare two variables to memorize the opposite edges in case
    // of tetrahedron
    double oppEdgesThat[3][3];
    double oppEdgesTho[3][3];

    // Declare a variable to memorize the speed of tho relative to that
    double relSpeed[3];
    relSpeed[0] = tho->speed[0] - that->speed[0];
    relSpeed[1] = tho->speed[1] - that->speed[1];
    relSpeed[2] = tho->speed[2] - that->speed[2];

    // Declare two variables to memorize the number of edges, by default 3
    int nbEdgesThat = 3;
    int nbEdgesTho = 3;

    // If the first Frame is a tetrahedron
    if (that->type == FrameTetrahedron) {

        // Shortcuts
        const double* frameCompA = that->comp[0];
        const double* frameCompB = that->comp[1];
        const double* frameCompC = that->comp[2];

        // Initialise the opposite edges
        oppEdgesThat[0][0] = frameCompB[0] - frameCompA[0];
        oppEdgesThat[0][1] = frameCompB[1] - frameCompA[1];
        oppEdgesThat[0][2] = frameCompB[2] - frameCompA[2];

        oppEdgesThat[1][0] = frameCompB[0] - frameCompC[0];
        oppEdgesThat[1][1] = frameCompB[1] - frameCompC[1];
        oppEdgesThat[1][2] = frameCompB[2] - frameCompC[2];

        oppEdgesThat[2][0] = frameCompC[0] - frameCompA[0];
        oppEdgesThat[2][1] = frameCompC[1] - frameCompA[1];
        oppEdgesThat[2][2] = frameCompC[2] - frameCompA[2];

        // Correct the number of edges
        nbEdgesThat = 6;
    }

    // If the second Frame is a tetrahedron

```

```

if (tho->type == FrameTetrahedron) {

    // Shortcuts
    const double* frameCompA = tho->comp[0];
    const double* frameCompB = tho->comp[1];
    const double* frameCompC = tho->comp[2];

    // Initialise the opposite edges
    oppEdgesTho[0][0] = frameCompB[0] - frameCompA[0];
    oppEdgesTho[0][1] = frameCompB[1] - frameCompA[1];
    oppEdgesTho[0][2] = frameCompB[2] - frameCompA[2];

    oppEdgesTho[1][0] = frameCompB[0] - frameCompC[0];
    oppEdgesTho[1][1] = frameCompB[1] - frameCompC[1];
    oppEdgesTho[1][2] = frameCompB[2] - frameCompC[2];

    oppEdgesTho[2][0] = frameCompC[0] - frameCompA[0];
    oppEdgesTho[2][1] = frameCompC[1] - frameCompA[1];
    oppEdgesTho[2][2] = frameCompC[2] - frameCompA[2];

    // Correct the number of edges
    nbEdgesTho = 6;
}

// Declare variables to loop on Frames and commonalize code
const Frame3DTime* frame = that;
const double (*oppEdgesA)[3] = oppEdgesThat;

// Loop to commonalize code when checking SAT based on that's edges
// and then tho's edges
for (int iFrame = 2;
     iFrame--;) {

    // Shortcuts
    FrameType frameType = frame->type;
    const double* frameCompA = frame->comp[0];
    const double* frameCompB = frame->comp[1];
    const double* frameCompC = frame->comp[2];

    // Declare a variable to memorize the number of faces, by default 3
    int nbFaces = 3;

    // Declare a variable to memorize the normal to faces
    // Arrangement is normFaces[iFace][iAxis]
    double normFaces[10][3];

    // Initialise the normal to faces
    normFaces[0][0] =
        frameCompA[1] * frameCompB[2] -
        frameCompA[2] * frameCompB[1];
    normFaces[0][1] =
        frameCompA[2] * frameCompB[0] -
        frameCompA[0] * frameCompB[2];
    normFaces[0][2] =
        frameCompA[0] * frameCompB[1] -
        frameCompA[1] * frameCompB[0];

    normFaces[1][0] =
        frameCompA[1] * frameCompC[2] -
        frameCompA[2] * frameCompC[1];
    normFaces[1][1] =

```

```

        frameCompA[2] * frameCompC[0] -
        frameCompA[0] * frameCompC[2];
normFaces[1][2] =
        frameCompA[0] * frameCompC[1] -
        frameCompA[1] * frameCompC[0];

normFaces[2][0] =
        frameCompC[1] * frameCompB[2] -
        frameCompC[2] * frameCompB[1];
normFaces[2][1] =
        frameCompC[2] * frameCompB[0] -
        frameCompC[0] * frameCompB[2];
normFaces[2][2] =
        frameCompC[0] * frameCompB[1] -
        frameCompC[1] * frameCompB[0];

// If the frame is a tetrahedron
if (frameType == FrameTetrahedron) {

    // Shortcuts
    const double* oppEdgeA = oppEdgesA[0];
    const double* oppEdgeB = oppEdgesA[1];

    // Initialise the normal to the opposite face
    normFaces[3][0] =
        oppEdgeA[1] * oppEdgeB[2] -
        oppEdgeA[2] * oppEdgeB[1];
    normFaces[3][1] =
        oppEdgeA[2] * oppEdgeB[0] -
        oppEdgeA[0] * oppEdgeB[2];
    normFaces[3][2] =
        oppEdgeA[0] * oppEdgeB[1] -
        oppEdgeA[1] * oppEdgeB[0];

    // Correct the number of faces
    nbFaces = 4;
}

// If we are checking the frame 'tho'
if (frame == tho) {

    // Add the normal to the virtual faces created by the speed
    // of tho relative to that

    normFaces[nbFaces][0] =
        relSpeed[1] * frameCompA[2] -
        relSpeed[2] * frameCompA[1];
    normFaces[nbFaces][1] =
        relSpeed[2] * frameCompA[0] -
        relSpeed[0] * frameCompA[2];
    normFaces[nbFaces][2] =
        relSpeed[0] * frameCompA[1] -
        relSpeed[1] * frameCompA[0];
    if (fabs(normFaces[nbFaces][0]) > EPSILON ||
        fabs(normFaces[nbFaces][1]) > EPSILON ||
        fabs(normFaces[nbFaces][2]) > EPSILON)
        ++nbFaces;

    normFaces[nbFaces][0] =
        relSpeed[1] * frameCompB[2] -
        relSpeed[2] * frameCompB[1];

```



```

normFaces[nbFaces][1] =
    relSpeed[2] * frameCompB[0] -
    relSpeed[0] * frameCompB[2];
normFaces[nbFaces][2] =
    relSpeed[0] * frameCompB[1] -
    relSpeed[1] * frameCompB[0];
if (fabs(normFaces[nbFaces][0]) > EPSILON ||
    fabs(normFaces[nbFaces][1]) > EPSILON ||
    fabs(normFaces[nbFaces][2]) > EPSILON)
    ++nbFaces;

normFaces[nbFaces][0] =
    relSpeed[1] * frameCompC[2] -
    relSpeed[2] * frameCompC[1];
normFaces[nbFaces][1] =
    relSpeed[2] * frameCompC[0] -
    relSpeed[0] * frameCompC[2];
normFaces[nbFaces][2] =
    relSpeed[0] * frameCompC[1] -
    relSpeed[1] * frameCompC[0];
if (fabs(normFaces[nbFaces][0]) > EPSILON ||
    fabs(normFaces[nbFaces][1]) > EPSILON ||
    fabs(normFaces[nbFaces][2]) > EPSILON)
    ++nbFaces;

if (frameType == FrameTetrahedron) {

    const double* oppEdgeA = oppEdgesA[0];
    const double* oppEdgeB = oppEdgesA[1];
    const double* oppEdgeC = oppEdgesA[2];

    normFaces[nbFaces][0] =
        relSpeed[1] * oppEdgeA[2] -
        relSpeed[2] * oppEdgeA[1];
    normFaces[nbFaces][1] =
        relSpeed[2] * oppEdgeA[0] -
        relSpeed[0] * oppEdgeA[2];
    normFaces[nbFaces][2] =
        relSpeed[0] * oppEdgeA[1] -
        relSpeed[1] * oppEdgeA[0];
    if (fabs(normFaces[nbFaces][0]) > EPSILON ||
        fabs(normFaces[nbFaces][1]) > EPSILON ||
        fabs(normFaces[nbFaces][2]) > EPSILON)
        ++nbFaces;

    normFaces[nbFaces][0] =
        relSpeed[1] * oppEdgeB[2] -
        relSpeed[2] * oppEdgeB[1];
    normFaces[nbFaces][1] =
        relSpeed[2] * oppEdgeB[0] -
        relSpeed[0] * oppEdgeB[2];
    normFaces[nbFaces][2] =
        relSpeed[0] * oppEdgeB[1] -
        relSpeed[1] * oppEdgeB[0];
    if (fabs(normFaces[nbFaces][0]) > EPSILON ||
        fabs(normFaces[nbFaces][1]) > EPSILON ||
        fabs(normFaces[nbFaces][2]) > EPSILON)
        ++nbFaces;

    normFaces[nbFaces][0] =
        relSpeed[1] * oppEdgeC[2] -
        relSpeed[2] * oppEdgeC[1];

```

```

        normFaces[nbFaces][1] =
            relSpeed[2] * oppEdgeC[0] -
            relSpeed[0] * oppEdgeC[2];
        normFaces[nbFaces][2] =
            relSpeed[0] * oppEdgeC[1] -
            relSpeed[1] * oppEdgeC[0];
        if (fabs(normFaces[nbFaces][0]) > EPSILON ||
            fabs(normFaces[nbFaces][1]) > EPSILON ||
            fabs(normFaces[nbFaces][2]) > EPSILON)
            ++nbFaces;
    }
}

// Loop on the frame's faces
for (int iFace = nbFaces;
     iFace--;) {

    // Check against the current face's normal
    bool isIntersection =
        CheckAxis3DTime(
            that,
            tho,
            normFaces[iFace],
            relSpeed);

    // If the axis is separating the Frames
    if (isIntersection == false) {

        // The Frames are not in intersection,
        // terminate the test
        return false;
    }
}

// Switch the frame to test against the second Frame
frame = tho;
oppEdgesA = oppEdgesTho;
}

// Loop on the pair of edges between the two frames
for (int iEdgeThat = nbEdgesThat;
     iEdgeThat--;) {

    // Get the first edge
    const double* edgeThat =
        (iEdgeThat < 3 ?
         that->comp[iEdgeThat] :
         oppEdgesThat[iEdgeThat - 3]);

    for (int iEdgeTho = nbEdgesTho + 1;
         iEdgeTho--;) {

        // Get the second edge
        const double* edgeTho =
            (iEdgeTho == nbEdgesTho ?
             relSpeed :
             (iEdgeTho < 3 ?
              tho->comp[iEdgeTho] :

```

```

        oppEdgesTho[iEdgeTho - 3]));

// Get the cross product of the two edges
double axis[3];
axis[0] = edgeThat[1] * edgeTho[2] - edgeThat[2] * edgeTho[1];
axis[1] = edgeThat[2] * edgeTho[0] - edgeThat[0] * edgeTho[2];
axis[2] = edgeThat[0] * edgeTho[1] - edgeThat[1] * edgeTho[0];

// Check against the cross product of the two edges
bool isIntersection =
    CheckAxis3DTime(
        that,
        tho,
        axis,
        relSpeed);

// If the axis is separating the Frames
if (isIntersection == false) {

    // The Frames are not in intersection,
    // terminate the test
    return false;

}

}

}

// If we reaches here, it means the two Frames are intersecting
return true;

}

// Check the intersection constraint for Frames that and tho
// relatively to axis
bool CheckAxis3D(
    const Frame3D* const that,
    const Frame3D* const tho,
    const double* const axis) {

    // Declare variables to memorize the boundaries of projection
    // of the two frames on the current edge
    double bdgBoxA[2];
    double bdgBoxB[2];

    // Declare two variables to loop on Frames and commonalize code
    const Frame3D* frame = that;
    double* bdgBox = bdgBoxA;

    // Loop on Frames
    for (int iFrame = 2;
        iFrame--;) {

        // Shortcuts
        const double* frameOrig = frame->orig;
        const double* frameCompA = frame->comp[0];
        const double* frameCompB = frame->comp[1];
        const double* frameCompC = frame->comp[2];
        FrameType frameType = frame->type;

        // Get the number of vertices of frame

```

```

int nbVertices = (frameType == FrameTetrahedron ? 4 : 8);

// Declare a variable to memorize if the current vertex is
// the first in the loop, used to initialize the boundaries
bool firstVertex = true;

// Loop on vertices of the frame
for (int iVertex = nbVertices;
     iVertex--;) {

    // Get the vertex
    double vertex[3];
    vertex[0] = frameOrig[0];
    vertex[1] = frameOrig[1];
    vertex[2] = frameOrig[2];
    switch (iVertex) {
        case 7:
            vertex[0] +=
                frameCompA[0] + frameCompB[0] + frameCompC[0];
            vertex[1] +=
                frameCompA[1] + frameCompB[1] + frameCompC[1];
            vertex[2] +=
                frameCompA[2] + frameCompB[2] + frameCompC[2];
            break;
        case 6:
            vertex[0] += frameCompB[0] + frameCompC[0];
            vertex[1] += frameCompB[1] + frameCompC[1];
            vertex[2] += frameCompB[2] + frameCompC[2];
            break;
        case 5:
            vertex[0] += frameCompA[0] + frameCompC[0];
            vertex[1] += frameCompA[1] + frameCompC[1];
            vertex[2] += frameCompA[2] + frameCompC[2];
            break;
        case 4:
            vertex[0] += frameCompA[0] + frameCompB[0];
            vertex[1] += frameCompA[1] + frameCompB[1];
            vertex[2] += frameCompA[2] + frameCompB[2];
            break;
        case 3:
            vertex[0] += frameCompC[0];
            vertex[1] += frameCompC[1];
            vertex[2] += frameCompC[2];
            break;
        case 2:
            vertex[0] += frameCompB[0];
            vertex[1] += frameCompB[1];
            vertex[2] += frameCompB[2];
            break;
        case 1:
            vertex[0] += frameCompA[0];
            vertex[1] += frameCompA[1];
            vertex[2] += frameCompA[2];
            break;
        default:
            break;
    }

    // Get the projection of the vertex on the axis
    double proj =
        vertex[0] * axis[0] +
        vertex[1] * axis[1] +

```

```

        vertex[2] * axis[2];

// If it's the first vertex
if (firstVertex == true) {

    // Initialize the boundaries of the projection of the
    // Frame on the edge
    bdgBox[0] = proj;
    bdgBox[1] = proj;

    // Update the flag to memorize we did the first vertex
    firstVertex = false;

// Else, it's not the first vertex
} else {

    // Update the boundaries of the projection of the Frame on
    // the edge
    if (bdgBox[0] > proj)
        bdgBox[0] = proj;

    if (bdgBox[1] < proj)
        bdgBox[1] = proj;

}

}

// Switch the frame to check the vertices of the second Frame
frame = tho;
bdgBox = bdgBoxB;

}

// If the projections of the two frames on the edge are
// not intersecting
if (bdgBoxB[1] < bdgBoxA[0] ||
    bdgBoxA[1] < bdgBoxB[0]) {

    // There exists an axis which separates the Frames,
    // thus they are not in intersection
    return false;

}

// If we reaches here the two Frames are in intersection
return true;

}

// Check the intersection constraint for Frames that and tho
// relatively to axis
bool CheckAxis3DTime(
    const Frame3DTime* const that,
    const Frame3DTime* const tho,
    const double* const axis,
    const double* const relSpeed) {

    // Declare variables to memorize the boundaries of projection
    // of the two frames on the current edge
    double bdgBoxA[2];
    double bdgBoxB[2];

```

```

// Declare two variables to loop on Frames and commonalize code
const Frame3DTime* frame = that;
double* bdgBox = bdgBoxA;

// Loop on Frames
for (int iFrame = 2;
     iFrame--;) {

    // Shortcuts
    const double* frameOrig = frame->orig;
    const double* frameCompA = frame->comp[0];
    const double* frameCompB = frame->comp[1];
    const double* frameCompC = frame->comp[2];
    FrameType frameType = frame->type;

    // Get the number of vertices of frame
    int nbVertices = (frameType == FrameTetrahedron ? 4 : 8);

    // Declare a variable to memorize if the current vertex is
    // the first in the loop, used to initialize the boundaries
    bool firstVertex = true;

    // Loop on vertices of the frame
    for (int iVertex = nbVertices;
         iVertex--;) {

        // Get the vertex
        double vertex[3];
        vertex[0] = frameOrig[0];
        vertex[1] = frameOrig[1];
        vertex[2] = frameOrig[2];
        switch (iVertex) {
            case 7:
                vertex[0] +=
                    frameCompA[0] + frameCompB[0] + frameCompC[0];
                vertex[1] +=
                    frameCompA[1] + frameCompB[1] + frameCompC[1];
                vertex[2] +=
                    frameCompA[2] + frameCompB[2] + frameCompC[2];
                break;
            case 6:
                vertex[0] += frameCompB[0] + frameCompC[0];
                vertex[1] += frameCompB[1] + frameCompC[1];
                vertex[2] += frameCompB[2] + frameCompC[2];
                break;
            case 5:
                vertex[0] += frameCompA[0] + frameCompC[0];
                vertex[1] += frameCompA[1] + frameCompC[1];
                vertex[2] += frameCompA[2] + frameCompC[2];
                break;
            case 4:
                vertex[0] += frameCompA[0] + frameCompB[0];
                vertex[1] += frameCompA[1] + frameCompB[1];
                vertex[2] += frameCompA[2] + frameCompB[2];
                break;
            case 3:
                vertex[0] += frameCompC[0];
                vertex[1] += frameCompC[1];
                vertex[2] += frameCompC[2];
                break;
            case 2:

```

```

        vertex[0] += frameCompB[0];
        vertex[1] += frameCompB[1];
        vertex[2] += frameCompB[2];
        break;
    case 1:
        vertex[0] += frameCompA[0];
        vertex[1] += frameCompA[1];
        vertex[2] += frameCompA[2];
        break;
    default:
        break;
}

// Get the projection of the vertex on the axis
double proj =
    vertex[0] * axis[0] +
    vertex[1] * axis[1] +
    vertex[2] * axis[2];

// If it's the first vertex
if (firstVertex == true) {

    // Initialize the boundaries of the projection of the
    // Frame on the edge
    bdgBox[0] = proj;
    bdgBox[1] = proj;

    // Update the flag to memorize we did the first vertex
    firstVertex = false;

// Else, it's not the first vertex
} else {

    // Update the boundaries of the projection of the Frame on
    // the edge
    if (bdgBox[0] > proj)
        bdgBox[0] = proj;

    if (bdgBox[1] < proj)
        bdgBox[1] = proj;

}

// If we are checking the second frame's vertices
if (frame == tho) {

    // Check also the vertices moved by the relative speed
    vertex[0] += relSpeed[0];
    vertex[1] += relSpeed[1];
    vertex[2] += relSpeed[2];

    proj =
        vertex[0] * axis[0] +
        vertex[1] * axis[1] +
        vertex[2] * axis[2];

    if (bdgBox[0] > proj)
        bdgBox[0] = proj;

    if (bdgBox[1] < proj)
        bdgBox[1] = proj;
}

```

```

    }

}

// Switch the frame to check the vertices of the second Frame
frame = tho;
bdgBox = bdgBoxB;

}

// If the projections of the two frames on the edge are
// not intersecting
if (bdgBoxB[1] < bdgBoxA[0] ||
    bdgBoxA[1] < bdgBoxB[0]) {

    // There exists an axis which separates the Frames,
    // thus they are not in intersection
    return false;

}

// If we reaches here the two Frames are in intersection
return true;

}

```

10.3 Makefile

In this section I introduce the Makefile used to compile the code given in the previous sections. It also includes command used to run the unit tests, validation and qualification, and to generate the documentation.

```

COMPILER=gcc
OPTIMIZATION=-O3

all : compile run plot dynamicAnalysis 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; ./
    qualification; cd -

run3D:
    cd 3D; ./main > ../Results/main3D.txt; ./unitTests > ../Results/
    unitTests3D.txt; ./validation > ../Results/validation3D.txt; ./
    qualification; cd -

run2DTime:
    cd 2DTime; ./main > ../Results/main2DTime.txt; ./unitTests > ../
    Results/unitTests2DTime.txt; ./validation > ../Results/
    validation2DTime.txt; ./qualification; cd -

run3DTime:
    cd 3DTime; ./main > ../Results/main3DTime.txt; ./unitTests > ../
    Results/unitTests3DTime.txt; ./validation > ../Results/
    validation3DTime.txt; ./qualification; cd -

plot: cleanPlot plot2D plot2DTime plot3D plot3DTime

cleanPlot:
    rm -f Results/*.png

plot2D:
    cd Results; gnuplot qualification2D.gnu; cd -

plot2DTime:
    cd Results; gnuplot qualification2DTime.gnu; cd -

plot3D:
    cd Results; gnuplot qualification3D.gnu; cd -

plot3DTime:
    cd Results; gnuplot qualification3DTime.gnu; cd -

doc:
    cd Doc; make latex; cd -

getRuntimeEnvironment:
    echo "uname -v\n" > runtimeEnv.txt; uname -v >> runtimeEnv.txt; echo
    "\n=====\n" >> runtimeEnv.txt; echo "lshw -short\n" >>
    runtimeEnv.txt; sudo lshw -short >> runtimeEnv.txt; echo "\n
    ====\n" >> runtimeEnv.txt; echo "lscpu\n" >> runtimeEnv
    .txt; lscpu >> runtimeEnv.txt; echo "\n=====\n" >>
    runtimeEnv.txt; echo "$ (COMPILER) -v\n" >> runtimeEnv.txt; $(
    COMPILER) -v 1>> runtimeEnv.txt 2>> runtimeEnv.txt

dynamicAnalysis:
    make valgrind 1> dynamicAnalysis.txt 2> dynamicAnalysis.txt

```

10.3.1 2D static

```
all : main unitTests validation qualification

COMPILER?=gcc
OPTIMIZATION?=-O3
BUILD_ARG=$(OPTIMIZATION) -I../SAT -I../Frame

main : main.o fmb2d.o frame.o Makefile
      $(COMPILER) -o main main.o fmb2d.o frame.o

main.o : main.c fmb2d.h ../Frame/frame.h Makefile
      $(COMPILER) -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb2d.o frame.o Makefile
      $(COMPILER) -o unitTests unitTests.o fmb2d.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb2d.h ../Frame/frame.h Makefile
      $(COMPILER) -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb2d.o sat.o frame.o Makefile
      $(COMPILER) -o validation validation.o fmb2d.o sat.o frame.o

validation.o : validation.c fmb2d.h ../SAT/sat.h ../Frame/frame.h Makefile
      $(COMPILER) -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb2d.o sat.o frame.o Makefile
      $(COMPILER) -o qualification qualification.o fmb2d.o sat.o frame.o $(LINK_ARG)

qualification.o : qualification.c fmb2d.h ../SAT/sat.h ../Frame/frame.h
      Makefile
      $(COMPILER) -c qualification.c $(BUILD_ARG)

fmb2d.o : fmb2d.c fmb2d.h ../Frame/frame.h Makefile
      $(COMPILER) -c fmb2d.c $(BUILD_ARG)

sat.o : ../SAT/sat.c ../SAT/sat.h ../Frame/frame.h Makefile
      $(COMPILER) -c ../SAT/sat.c $(BUILD_ARG)

frame.o : ../Frame/frame.c ../Frame/frame.h Makefile
      $(COMPILER) -c ../Frame/frame.c $(BUILD_ARG)

clean :
      rm -f *.o main unitTests validation qualification

valgrind :
      valgrind -v --track-origins=yes --leak-check=full \
      --gen-suppressions=yes --show-leak-kinds=all ./unitTests
```

10.3.2 3D static

```
all : main unitTests validation qualification

COMPILER?=gcc
OPTIMIZATION?=-O3
BUILD_ARG=$(OPTIMIZATION) -I../SAT -I../Frame

main : main.o fmb3d.o frame.o Makefile
      $(COMPILER) -o main main.o fmb3d.o frame.o
```

```

main.o : main.c fmb3d.h ../Frame/frame.h Makefile
$(COMPILER) -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb3d.o frame.o Makefile
$(COMPILER) -o unitTests unitTests.o fmb3d.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb3d.h ../Frame/frame.h Makefile
$(COMPILER) -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb3d.o sat.o frame.o Makefile
$(COMPILER) -o validation validation.o fmb3d.o sat.o frame.o

validation.o : validation.c fmb3d.h ../SAT/sat.h ../Frame/frame.h Makefile
$(COMPILER) -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb3d.o sat.o frame.o Makefile
$(COMPILER) -o qualification qualification.o fmb3d.o sat.o frame.o $(LINK_ARG)

qualification.o : qualification.c fmb3d.h ../SAT/sat.h ../Frame/frame.h
Makefile
$(COMPILER) -c qualification.c $(BUILD_ARG)

fmb3d.o : fmb3d.c fmb3d.h ../Frame/frame.h Makefile
$(COMPILER) -c fmb3d.c $(BUILD_ARG)

sat.o : ../SAT/sat.c ../SAT/sat.h ../Frame/frame.h Makefile
$(COMPILER) -c ../SAT/sat.c $(BUILD_ARG)

frame.o : ../Frame/frame.c ../Frame/frame.h Makefile
$(COMPILER) -c ../Frame/frame.c $(BUILD_ARG)

clean :
rm -f *.o main unitTests validation qualification

valgrind :
valgrind -v --track-origins=yes --leak-check=full \
--gen-suppressions=yes --show-leak-kinds=all ./unitTests

```

10.3.3 2D dynamic

```

all : main unitTests validation qualification

COMPILER?=gcc
OPTIMIZATION?=-O3
BUILD_ARG=$(OPTIMIZATION) -I../SAT -I../Frame

main : main.o fmb2dt.o frame.o Makefile
$(COMPILER) -o main main.o fmb2dt.o frame.o

main.o : main.c fmb2dt.h ../Frame/frame.h Makefile
$(COMPILER) -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb2dt.o frame.o Makefile
$(COMPILER) -o unitTests unitTests.o fmb2dt.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb2dt.h ../Frame/frame.h Makefile
$(COMPILER) -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb2dt.o sat.o frame.o Makefile
$(COMPILER) -o validation validation.o fmb2dt.o sat.o frame.o

```

```

validation.o : validation.c fmb2dt.h ../SAT/sat.h ../Frame/frame.h Makefile
                $(COMPILER) -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb2dt.o sat.o frame.o Makefile
                $(COMPILER) -o qualification qualification.o fmb2dt.o sat.o frame.o
                $(LINK_ARG)

qualification.o : qualification.c fmb2dt.h ../SAT/sat.h ../Frame/frame.h
                Makefile
                $(COMPILER) -c qualification.c $(BUILD_ARG)

fmb2dt.o : fmb2dt.c fmb2dt.h ../Frame/frame.h Makefile
                $(COMPILER) -c fmb2dt.c $(BUILD_ARG)

sat.o : ../SAT/sat.c ../SAT/sat.h ../Frame/frame.h Makefile
                $(COMPILER) -c ../SAT/sat.c $(BUILD_ARG)

frame.o : ../Frame/frame.c ../Frame/frame.h Makefile
                $(COMPILER) -c ../Frame/frame.c $(BUILD_ARG)

clean :
        rm -f *.o main unitTests validation qualification

valgrind :
        valgrind -v --track-origins=yes --leak-check=full \
        --gen-suppressions=yes --show-leak-kinds=all ./unitTests

```

10.3.4 3D dynamic

```

all : main unitTests validation qualification

COMPILER?=gcc
OPTIMIZATION?=-O3
BUILD_ARG=$(OPTIMIZATION) -I../SAT -I../Frame

main : main.o fmb3dt.o frame.o Makefile
        $(COMPILER) -o main main.o fmb3dt.o frame.o

main.o : main.c fmb3dt.h ../Frame/frame.h Makefile
        $(COMPILER) -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb3dt.o frame.o Makefile
        $(COMPILER) -o unitTests unitTests.o fmb3dt.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb3dt.h ../Frame/frame.h Makefile
        $(COMPILER) -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb3dt.o sat.o frame.o Makefile
        $(COMPILER) -o validation validation.o fmb3dt.o sat.o frame.o

validation.o : validation.c fmb3dt.h ../SAT/sat.h ../Frame/frame.h Makefile
        $(COMPILER) -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb3dt.o sat.o frame.o Makefile
        $(COMPILER) -o qualification qualification.o fmb3dt.o sat.o frame.o
        $(LINK_ARG)

qualification.o : qualification.c fmb3dt.h ../SAT/sat.h ../Frame/frame.h
                Makefile
                $(COMPILER) -c qualification.c $(BUILD_ARG)

```

```

fmb3dt.o : fmb3dt.c fmb3dt.h ../Frame/frame.h Makefile
$(COMPILER) -c fmb3dt.c $(BUILD_ARG)

sat.o : ../SAT/sat.c ../SAT/sat.h ../Frame/frame.h Makefile
$(COMPILER) -c ../SAT/sat.c $(BUILD_ARG)

frame.o : ../Frame/frame.c ../Frame/frame.h Makefile
$(COMPILER) -c ../Frame/frame.c $(BUILD_ARG)

clean :
    rm -f *.o main unitTests validation qualification

valgrind :
    valgrind -v --track-origins=yes --leak-check=full \
    --gen-suppressions=yes --show-leak-kinds=all ./unitTests

```

10.3.5 Doc

```

latex:
    pdflatex -synctex=1 -interaction=nonstopmode -shell-escape fmb.tex

```

10.4 Dynamic analysis

```

make[1]: Entering directory '/home/bayashi/GitHub/FMB'
cd 2D; make valgrind; cd -
make[2]: Entering directory '/home/bayashi/GitHub/FMB/2D'
valgrind -v --track-origins=yes --leak-check=full \
--gen-suppressions=yes --show-leak-kinds=all ./unitTests
==4492== Memcheck, a memory error detector
==4492== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4492== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4492== Command: ./unitTests
==4492==
--4492-- Valgrind options:
--4492--      -v
--4492--      --track-origins=yes
--4492--      --leak-check=full
--4492--      --gen-suppressions=yes
--4492--      --show-leak-kinds=all
--4492-- Contents of /proc/version:
--4492--   Linux version 4.4.0-170-generic (buildd@lgw01-amd64-015) (gcc
        version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12) ) #199-Ubuntu
        SMP Thu Nov 14 01:44:41 UTC 2019
--4492--
--4492-- Arch and hwcaps: X86, LittleEndian, x86-mmext-sse1-sse2-sse3
--4492-- Page sizes: currently 4096, max supported 4096
--4492-- Valgrind library directory: /usr/lib/valgrind
--4492-- Reading syms from /lib/i386-linux-gnu/ld-2.23.so
--4492--   Considering /lib/i386-linux-gnu/ld-2.23.so ..
--4492--   .. CRC mismatch (computed f6bf32f6 wanted cd116314)
--4492--   Considering /usr/lib/debug/lib/i386-linux-gnu/ld-2.23.so ..
--4492--   .. CRC is valid
--4492-- Reading syms from /home/bayashi/GitHub/FMB/2D/unitTests
--4492-- Reading syms from /usr/lib/valgrind/memcheck-x86-linux
--4492--   Considering /usr/lib/valgrind/memcheck-x86-linux ..
--4492--   .. CRC mismatch (computed 6036da3f wanted 652e483f)
--4492--   object doesn't have a symbol table
--4492--   object doesn't have a dynamic symbol table
--4492-- Scheduler: using generic scheduler lock implementation.

```

```

--4492-- Reading suppressions file: /usr/lib/valgrind/default.supp
==4492== embedded gdbserver: reading from /tmp/vgdb-pipe-from-vgdb-to-4492-
by-bayashi-on-???
==4492== embedded gdbserver: writing to /tmp/vgdb-pipe-to-vgdb-from-4492-
by-bayashi-on-???
==4492== embedded gdbserver: shared mem /tmp/vgdb-pipe-shared-mem-vgdb
-4492-by-bayashi-on-???
==4492==
==4492== TO CONTROL THIS PROCESS USING vgdb (which you probably
==4492== don't want to do, unless you know exactly what you're doing,
==4492== or are doing some strange experiment):
==4492== /usr/lib/valgrind/../../bin/vgdb --pid=4492 ...command...
==4492==
==4492== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==4492== /path/to/gdb ./unitTests
==4492== and then give GDB the following command
==4492== target remote | /usr/lib/valgrind/../../bin/vgdb --pid=4492
==4492== --pid is optional if only one valgrind process is running
==4492==
--4492-- REDIR: 0x4019030 (ld-linux.so.2:strlen) redirected to 0x380490f2
(???)
--4492-- REDIR: 0x4018df0 (ld-linux.so.2:index) redirected to 0x380490cd
(???)
--4492-- Reading syms from /usr/lib/valgrind/vgpreload_core-x86-linux.so
--4492-- Considering /usr/lib/valgrind/vgpreload_core-x86-linux.so ..
--4492-- .. CRC mismatch (computed 5b95c0c4 wanted 39e79efa)
--4492-- object doesn't have a symbol table
--4492-- Reading syms from /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so
--4492-- Considering /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so ..
--4492-- .. CRC mismatch (computed 4d5a1526 wanted 33571d5b)
--4492-- object doesn't have a symbol table
==4492== WARNING: new redirection conflicts with existing -- ignoring it
--4492-- old: 0x04019030 (strlen ) R-> (0000.0) 0x380490f2
???
--4492-- new: 0x04019030 (strlen ) R-> (2007.0) 0x0402f410
strlen
--4492-- Reading syms from /lib/i386-linux-gnu/libc-2.23.so
--4492-- Considering /lib/i386-linux-gnu/libc-2.23.so ..
--4492-- .. CRC mismatch (computed 05488cfc wanted a5215580)
--4492-- Considering /usr/lib/debug/lib/i386-linux-gnu/libc-2.23.so ..
--4492-- .. CRC is valid
--4492-- REDIR: 0x40d34b0 (libc.so.6:strncasecmp) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4492-- REDIR: 0x40da140 (libc.so.6:memrchr) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4492-- REDIR: 0x40ecb20 (libc.so.6:wcslen) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4492-- REDIR: 0x40d17e0 (libc.so.6:__GI_strrchr) redirected to 0x402ee00 (
__GI_strrchr)
--4492-- REDIR: 0x40ccf00 (libc.so.6:malloc) redirected to 0x402c110 (malloc
)
--4492-- REDIR: 0x40d5280 (libc.so.6:strchrnul) redirected to 0x40330b0 (
strchrnul)
--4492-- REDIR: 0x40d30d0 (libc.so.6:__GI_memcpy) redirected to 0x40332c0 (
__GI_memcpy)
--4492-- REDIR: 0x40d15c0 (libc.so.6:__GI_strlen) redirected to 0x402f390 (
__GI_strlen)
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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

```

```

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.000000)

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 (no inter)

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 (no inter)

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
minXY(0.250000,0.000000)-maxXY(0.750000,1.000000)

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
minXY(0.250000,0.000000)-maxXY(0.750000,1.000000)

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
minXY(0.000000,0.250000)-maxXY(1.000000,0.750000)

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
minXY(0.000000,0.250000)-maxXY(1.000000,0.750000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(0.500000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(0.500000,1.000000)

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
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(1.500000,0.000000)-maxXY(1.666667,1.000000)

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
minXY(1.500000,0.500000)-maxXY(2.000000,1.000000)

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
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

```

```

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.500000)

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.500000)

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,0.500000)-maxXY(1.000000,1.000000)

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
minXY(0.500000,1.000000)-maxXY(1.000000,1.500000)

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
minXY(0.500000,1.000000)-maxXY(1.000000,1.500000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,0.500000)

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
minXY(0.000000,0.000000)-maxXY(1.000000,0.500000)

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
minXY(0.000000,0.000000)-maxXY(0.500000,0.500000)

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
minXY(0.000000,0.000000)-maxXY(0.500000,0.500000)

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 (no inter)

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 (no inter)

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
minXY(0.000000,0.500000)-maxXY(1.000000,1.000000)

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
minXY(0.000000,0.500000)-maxXY(1.000000,1.000000)

To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
against
To(1.010000,1.010000) x(-1.000000,0.000000) y(0.000000,-1.000000)
Succeed (no inter)

To(1.010000,1.010000) x(-1.000000,0.000000) y(0.000000,-1.000000)
against
To(0.000000,0.000000) x(1.000000,0.000000) y(0.000000,1.000000)
Succeed (no inter)

To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
against
To(1.000000,1.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
Succeed
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

To(1.000000,1.000000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
against
To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
Succeed
minXY(0.500000,0.000000)-maxXY(1.000000,1.000000)

To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
against
To(1.010000,1.500000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
Succeed (no inter)

To(1.010000,1.500000) x(-0.500000,-0.500000) y(0.000000,-1.000000)
against

```

```

To(0.000000,0.000000) x(1.000000,0.500000) y(0.500000,1.000000)
  Succeed (no inter)

All unit tests 2D have succeed.
--4492-- REDIR: 0x40cd470 (libc.so.6:free) redirected to 0x402d2f0 (free)
==4492==
==4492== HEAP SUMMARY:
==4492==     in use at exit: 0 bytes in 0 blocks
==4492==   total heap usage: 1 allocs, 1 frees, 4,096 bytes allocated
==4492==
==4492== All heap blocks were freed -- no leaks are possible
==4492==
==4492== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4492== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
make[2]: Leaving directory '/home/bayashi/GitHub/FMB/2D'
/home/bayashi/GitHub/FMB
cd 2DTime; make valgrind; cd -
make[2]: Entering directory '/home/bayashi/GitHub/FMB/2DTime'
valgrind -v --track-origins=yes --leak-check=full \
--gen-suppressions=yes --show-leak-kinds=all ./unitTests
==4495== Memcheck, a memory error detector
==4495== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4495== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4495== Command: ./unitTests
==4495==
--4495-- Valgrind options:
--4495--     -v
--4495--     --track-origins=yes
--4495--     --leak-check=full
--4495--     --gen-suppressions=yes
--4495--     --show-leak-kinds=all
--4495-- Contents of /proc/version:
--4495--   Linux version 4.4.0-170-generic (buildd@lgw01-amd64-015) (gcc
        version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12) ) #199-Ubuntu
        SMP Thu Nov 14 01:44:41 UTC 2019
--4495--
--4495-- Arch and hwcaps: X86, LittleEndian, x86-mmext-sse1-sse2-sse3
--4495-- Page sizes: currently 4096, max supported 4096
--4495-- Valgrind library directory: /usr/lib/valgrind
--4495-- Reading syms from /lib/i386-linux-gnu/ld-2.23.so
--4495--   Considering /lib/i386-linux-gnu/ld-2.23.so ..
--4495--   .. CRC mismatch (computed f6bf32f6 wanted cd116314)
--4495--   Considering /usr/lib/debug/lib/i386-linux-gnu/ld-2.23.so ..
--4495--   .. CRC is valid
--4495-- Reading syms from /home/bayashi/GitHub/FMB/2DTime/unitTests
--4495-- Reading syms from /usr/lib/valgrind/memcheck-x86-linux
--4495--   Considering /usr/lib/valgrind/memcheck-x86-linux ..
--4495--   .. CRC mismatch (computed 6036da3f wanted 652e483f)
--4495--   object doesn't have a symbol table
--4495--   object doesn't have a dynamic symbol table
--4495-- Scheduler: using generic scheduler lock implementation.
--4495-- Reading suppressions file: /usr/lib/valgrind/default.supp
==4495== embedded gdbserver: reading from /tmp/vgdb-pipe-from-vgdb-to-4495-
        by-bayashi-on-???
==4495== embedded gdbserver: writing to   /tmp/vgdb-pipe-to-vgdb-from-4495-
        by-bayashi-on-???
==4495== embedded gdbserver: shared mem   /tmp/vgdb-pipe-shared-mem-vgdb
        -4495-by-bayashi-on-???
==4495==
==4495== TO CONTROL THIS PROCESS USING vgdb (which you probably
==4495== don't want to do, unless you know exactly what you're doing,
==4495== or are doing some strange experiment):

```

```

==4495== /usr/lib/valgrind/../../bin/vgdb --pid=4495 ...command...
==4495==
==4495== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==4495== /path/to/gdb ./unitTests
==4495== and then give GDB the following command
==4495== target remote | /usr/lib/valgrind/../../bin/vgdb --pid=4495
==4495== --pid is optional if only one valgrind process is running
==4495==
--4495-- REDIR: 0x4019030 (ld-linux.so.2:strlen) redirected to 0x380490f2
(???)
--4495-- REDIR: 0x4018df0 (ld-linux.so.2:index) redirected to 0x380490cd
(???)
--4495-- Reading syms from /usr/lib/valgrind/vgpreload_core-x86-linux.so
--4495-- Considering /usr/lib/valgrind/vgpreload_core-x86-linux.so ..
--4495-- .. CRC mismatch (computed 5b95c0c4 wanted 39e79efa)
--4495-- object doesn't have a symbol table
--4495-- Reading syms from /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so
--4495-- Considering /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so ..
--4495-- .. CRC mismatch (computed 4d5a1526 wanted 33571d5b)
--4495-- object doesn't have a symbol table
==4495== WARNING: new redirection conflicts with existing -- ignoring it
--4495-- old: 0x04019030 (strlen ) R-> (0000.0) 0x380490f2
???
--4495-- new: 0x04019030 (strlen ) R-> (2007.0) 0x0402f410
strlen
--4495-- Reading syms from /lib/i386-linux-gnu/libc-2.23.so
--4495-- Considering /lib/i386-linux-gnu/libc-2.23.so ..
--4495-- .. CRC mismatch (computed 05488cfc wanted a5215580)
--4495-- Considering /usr/lib/debug/lib/i386-linux-gnu/libc-2.23.so ..
--4495-- .. CRC is valid
--4495-- REDIR: 0x40d34b0 (libc.so.6:strncasecmp) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4495-- REDIR: 0x40da140 (libc.so.6:memrchr) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4495-- REDIR: 0x40ecb20 (libc.so.6:wcslen) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4495-- REDIR: 0x40d17e0 (libc.so.6:__GI_strrchr) redirected to 0x402ee00 (
__GI_strrchr)
--4495-- REDIR: 0x40ccf00 (libc.so.6:malloc) redirected to 0x402c110 (malloc
)
--4495-- REDIR: 0x40d5280 (libc.so.6:strchrnul) redirected to 0x40330b0 (
strchrnul)
--4495-- REDIR: 0x40d30d0 (libc.so.6:__GI_mempcpy) redirected to 0x40332c0 (
__GI_mempcpy)
--4495-- REDIR: 0x40d15c0 (libc.so.6:__GI_strlen) redirected to 0x402f390 (
__GI_strlen)
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 (no inter)

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 (no inter)

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)

```

```

against
Co(-1.010000,-1.010000) s(1.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed (no inter)

Co(-1.010000,-1.010000) s(1.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed (no inter)

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
minXYT(-1.000000,0.000000,0.000000)-maxXYT(2.000000,1.000000,1.000000)

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
minXYT(-1.000000,0.000000,0.000000)-maxXYT(1.000000,1.000000,1.000000)

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
against
Co(-1.000000,0.250000) s(4.000000,0.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
Succeed
minXYT(-1.500000,0.000000,0.125000)-maxXYT(2.500000,1.000000,0.500000)

Co(-1.000000,0.250000) s(4.000000,0.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed
minXYT(-0.500000,0.000000,0.125000)-maxXYT(1.500000,1.000000,0.500000)

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
against
Co(0.250000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
Succeed
minXYT(0.000000,-1.500000,0.125000)-maxXYT(1.000000,2.500000,0.500000)

Co(0.250000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed
minXYT(0.000000,-0.500000,0.125000)-maxXYT(1.000000,1.500000,0.500000)

Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)

```

```

against
Co(0.900000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
Succeed
minXYT(0.000000,-1.500000,0.125000)-maxXYT(1.400000,2.500000,0.500000)

Co(0.900000,-1.000000) s(0.000000,4.000000) x(0.500000,0.000000) y
(0.000000,0.500000)
against
Co(0.000000,0.000000) s(0.000000,0.000000) x(1.000000,0.000000) y
(0.000000,1.000000)
Succeed
minXYT(0.000000,-0.500000,0.125000)-maxXYT(1.400000,1.500000,0.500000)

All unit tests 2DTime have succeed.
--4495-- REDIR: 0x40cd470 (libc.so.6:free) redirected to 0x402d2f0 (free)
==4495==
==4495== HEAP SUMMARY:
==4495==      in use at exit: 0 bytes in 0 blocks
==4495==    total heap usage: 1 allocs, 1 frees, 4,096 bytes allocated
==4495==
==4495== All heap blocks were freed -- no leaks are possible
==4495==
==4495== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4495== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
make[2]: Leaving directory '/home/bayashi/GitHub/FMB/2DTime'
/home/bayashi/GitHub/FMB
cd 3D; make valgrind; cd -
make[2]: Entering directory '/home/bayashi/GitHub/FMB/3D'
valgrind -v --track-origins=yes --leak-check=full \
--gen-suppressions=yes --show-leak-kinds=all ./unitTests
==4498== Memcheck, a memory error detector
==4498== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4498== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4498== Command: ./unitTests
==4498==
--4498-- Valgrind options:
--4498--      -v
--4498--      --track-origins=yes
--4498--      --leak-check=full
--4498--      --gen-suppressions=yes
--4498--      --show-leak-kinds=all
--4498-- Contents of /proc/version:
--4498--   Linux version 4.4.0-170-generic (buildd@lgw01-amd64-015) (gcc
        version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12) ) #199-Ubuntu
        SMP Thu Nov 14 01:44:41 UTC 2019
--4498--
--4498-- Arch and hwcaps: X86, LittleEndian, x86-mmext-sse1-sse2-sse3
--4498-- Page sizes: currently 4096, max supported 4096
--4498-- Valgrind library directory: /usr/lib/valgrind
--4498-- Reading syms from /lib/i386-linux-gnu/ld-2.23.so
--4498--   Considering /lib/i386-linux-gnu/ld-2.23.so ..
--4498--   .. CRC mismatch (computed f6bf32f6 wanted cd116314)
--4498--   Considering /usr/lib/debug/lib/i386-linux-gnu/ld-2.23.so ..
--4498--   .. CRC is valid
--4498-- Reading syms from /home/bayashi/GitHub/FMB/3D/unitTests
--4498-- Reading syms from /usr/lib/valgrind/memcheck-x86-linux
--4498--   Considering /usr/lib/valgrind/memcheck-x86-linux ..
--4498--   .. CRC mismatch (computed 6036da3f wanted 652e483f)
--4498--   object doesn't have a symbol table
--4498--   object doesn't have a dynamic symbol table
--4498-- Scheduler: using generic scheduler lock implementation.

```

```

--4498-- Reading suppressions file: /usr/lib/valgrind/default.supp
==4498== embedded gdbserver: reading from /tmp/vgdb-pipe-from-vgdb-to-4498-
by-bayashi-on-???
==4498== embedded gdbserver: writing to /tmp/vgdb-pipe-to-vgdb-from-4498-
by-bayashi-on-???
==4498== embedded gdbserver: shared mem /tmp/vgdb-pipe-shared-mem-vgdb
-4498-by-bayashi-on-???
==4498==
==4498== TO CONTROL THIS PROCESS USING vgdb (which you probably
==4498== don't want to do, unless you know exactly what you're doing,
==4498== or are doing some strange experiment):
==4498== /usr/lib/valgrind/../../bin/vgdb --pid=4498 ...command...
==4498==
==4498== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==4498== /path/to/gdb ./unitTests
==4498== and then give GDB the following command
==4498== target remote | /usr/lib/valgrind/../../bin/vgdb --pid=4498
==4498== --pid is optional if only one valgrind process is running
==4498==
--4498-- REDIR: 0x4019030 (ld-linux.so.2:strlen) redirected to 0x380490f2
(???)
--4498-- REDIR: 0x4018df0 (ld-linux.so.2:index) redirected to 0x380490cd
(???)
--4498-- Reading syms from /usr/lib/valgrind/vgpreload_core-x86-linux.so
--4498-- Considering /usr/lib/valgrind/vgpreload_core-x86-linux.so ..
--4498-- .. CRC mismatch (computed 5b95c0c4 wanted 39e79efa)
--4498-- object doesn't have a symbol table
--4498-- Reading syms from /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so
--4498-- Considering /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so ..
--4498-- .. CRC mismatch (computed 4d5a1526 wanted 33571d5b)
--4498-- object doesn't have a symbol table
==4498== WARNING: new redirection conflicts with existing -- ignoring it
--4498-- old: 0x04019030 (strlen ) R-> (0000.0) 0x380490f2
???
--4498-- new: 0x04019030 (strlen ) R-> (2007.0) 0x0402f410
strlen
--4498-- Reading syms from /lib/i386-linux-gnu/libc-2.23.so
--4498-- Considering /lib/i386-linux-gnu/libc-2.23.so ..
--4498-- .. CRC mismatch (computed 05488cfc wanted a5215580)
--4498-- Considering /usr/lib/debug/lib/i386-linux-gnu/libc-2.23.so ..
--4498-- .. CRC is valid
--4498-- REDIR: 0x40d34b0 (libc.so.6:strncasecmp) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4498-- REDIR: 0x40da140 (libc.so.6:memrchr) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4498-- REDIR: 0x40ecb20 (libc.so.6:wcslen) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4498-- REDIR: 0x40d17e0 (libc.so.6:__GI_strrchr) redirected to 0x402ee00 (
__GI_strrchr)
--4498-- REDIR: 0x40ccf00 (libc.so.6:malloc) redirected to 0x402c110 (malloc
)
--4498-- REDIR: 0x40d5280 (libc.so.6:strchrnul) redirected to 0x40330b0 (
strchrnul)
--4498-- REDIR: 0x40d30d0 (libc.so.6:__GI_memcpy) redirected to 0x40332c0 (
__GI_memcpy)
--4498-- REDIR: 0x40d15c0 (libc.so.6:__GI_strlen) redirected to 0x402f390 (
__GI_strlen)
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)

```



```

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 (no inter)

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 (no inter)

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
minXYZ(0.500000,0.500000,-1.000000)-maxXYZ(1.000000,1.000000,-0.500000)

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
minXYZ(0.500000,0.500000,-1.000000)-maxXYZ(1.000000,1.000000,-0.500000)

Co(-1.010000,-1.010000,-1.010000) x(1.000000,0.000000,0.000000) y
(1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
(0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed (no inter)

Co(0.000000,0.000000,0.000000) x(1.000000,0.000000,0.000000) y
(0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(-1.010000,-1.010000,-1.010000) x(1.000000,0.000000,0.000000) y
(1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
Succeed (no inter)

Co(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
(1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
(0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed
minXYZ(0.000000,-0.500000,0.000000)-maxXYZ(1.000000,0.000000,1.000000)

Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
(0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
(1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
Succeed
minXYZ(0.000000,-0.500000,0.000000)-maxXYZ(1.000000,0.000000,1.000000)

To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
(1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
(0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)

```

```

Succeed (no inter)

Co(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
Succeed (no inter)

Co(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed
minXYZ(0.000000,-0.500000,0.000000)-maxXYZ(1.000000,0.000000,0.750000)

To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
Co(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
Succeed
minXYZ(0.000000,-0.500000,0.000000)-maxXYZ(1.000000,0.000000,1.000000)

To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed (no inter)

To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
To(-1.000000,-1.000000,-1.000000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
Succeed (no inter)

To(-0.500000,-1.000000,-0.500000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
against
To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
Succeed
minXYZ(0.000000,-0.500000,0.000000)-maxXYZ(0.500000,0.000000,0.500000)

To(0.000000,-0.500000,0.000000) x(1.000000,0.000000,0.000000) y
  (0.000000,1.000000,0.000000) z(0.000000,0.000000,1.000000)
against
To(-0.500000,-1.000000,-0.500000) x(1.000000,0.000000,0.000000) y
  (1.000000,1.000000,1.000000) z(0.000000,0.000000,1.000000)
Succeed
minXYZ(0.000000,-0.500000,0.000000)-maxXYZ(0.500000,0.000000,0.500000)

All unit tests 3D have succeed.
--4498-- REDIR: 0x40cd470 (libc.so.6:free) redirected to 0x402d2f0 (free)
==4498==
==4498== HEAP SUMMARY:
==4498==       in use at exit: 0 bytes in 0 blocks
==4498==   total heap usage: 1 allocs, 1 frees, 4,096 bytes allocated
==4498==

```

```

==4498== All heap blocks were freed -- no leaks are possible
==4498==
==4498== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4498== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
make[2]: Leaving directory '/home/bayashi/GitHub/FMB/3D'
/home/bayashi/GitHub/FMB
cd 3DTime; make valgrind; cd -
make[2]: Entering directory '/home/bayashi/GitHub/FMB/3DTime'
valgrind -v --track-origins=yes --leak-check=full \
--gen-suppressions=yes --show-leak-kinds=all ./unitTests
==4501== Memcheck, a memory error detector
==4501== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4501== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4501== Command: ./unitTests
==4501==
--4501-- Valgrind options:
--4501--     -v
--4501--     --track-origins=yes
--4501--     --leak-check=full
--4501--     --gen-suppressions=yes
--4501--     --show-leak-kinds=all
--4501-- Contents of /proc/version:
--4501--     Linux version 4.4.0-170-generic (buildd@lgw01-amd64-015) (gcc
        version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12) ) #199-Ubuntu
        SMP Thu Nov 14 01:44:41 UTC 2019
--4501--
--4501-- Arch and hwcaps: X86, LittleEndian, x86-mmxext-ssse3-sse2-sse3
--4501-- Page sizes: currently 4096, max supported 4096
--4501-- Valgrind library directory: /usr/lib/valgrind
--4501-- Reading syms from /lib/i386-linux-gnu/ld-2.23.so
--4501--   Considering /lib/i386-linux-gnu/ld-2.23.so ..
--4501--   .. CRC mismatch (computed f6bf32f6 wanted cd116314)
--4501--   Considering /usr/lib/debug/lib/i386-linux-gnu/ld-2.23.so ..
--4501--   .. CRC is valid
--4501-- Reading syms from /home/bayashi/GitHub/FMB/3DTime/unitTests
--4501-- Reading syms from /usr/lib/valgrind/memcheck-x86-linux
--4501--   Considering /usr/lib/valgrind/memcheck-x86-linux ..
--4501--   .. CRC mismatch (computed 6036da3f wanted 652e483f)
--4501--   object doesn't have a symbol table
--4501--   object doesn't have a dynamic symbol table
--4501-- Scheduler: using generic scheduler lock implementation.
--4501-- Reading suppressions file: /usr/lib/valgrind/default.supp
==4501== embedded gdbserver: reading from /tmp/vgdb-pipe-from-vgdb-to-4501-
        by-bayashi-on-???
==4501== embedded gdbserver: writing to   /tmp/vgdb-pipe-to-vgdb-from-4501-
        by-bayashi-on-???
==4501== embedded gdbserver: shared mem   /tmp/vgdb-pipe-shared-mem-vgdb
        -4501-by-bayashi-on-???
==4501==
==4501== TO CONTROL THIS PROCESS USING vgdb (which you probably
==4501== don't want to do, unless you know exactly what you're doing,
==4501== or are doing some strange experiment):
==4501==   /usr/lib/valgrind/../../bin/vgdb --pid=4501 ...command...
==4501==
==4501== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==4501==   /path/to/gdb ./unitTests
==4501== and then give GDB the following command
==4501==   target remote | /usr/lib/valgrind/../../bin/vgdb --pid=4501
==4501== --pid is optional if only one valgrind process is running
==4501==
--4501-- REDIR: 0x4019030 (ld-linux.so.2:strlen) redirected to 0x380490f2
        (???)

```

```

--4501-- REDIR: 0x4018df0 (ld-linux.so.2:index) redirected to 0x380490cd
(???)
--4501-- Reading syms from /usr/lib/valgrind/vgpreload_core-x86-linux.so
--4501-- Considering /usr/lib/valgrind/vgpreload_core-x86-linux.so ..
--4501-- .. CRC mismatch (computed 5b95c0c4 wanted 39e79efa)
--4501-- object doesn't have a symbol table
--4501-- Reading syms from /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so
--4501-- Considering /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so ..
--4501-- .. CRC mismatch (computed 4d5a1526 wanted 33571d5b)
--4501-- object doesn't have a symbol table
==4501== WARNING: new redirection conflicts with existing -- ignoring it
--4501-- old: 0x04019030 (strlen ) R-> (0000.0) 0x380490f2
???
--4501-- new: 0x04019030 (strlen ) R-> (2007.0) 0x0402f410
strlen
--4501-- Reading syms from /lib/i386-linux-gnu/libc-2.23.so
--4501-- Considering /lib/i386-linux-gnu/libc-2.23.so ..
--4501-- .. CRC mismatch (computed 05488cfc wanted a5215580)
--4501-- Considering /usr/lib/debug/lib/i386-linux-gnu/libc-2.23.so ..
--4501-- .. CRC is valid
--4501-- REDIR: 0x40d34b0 (libc.so.6:strncasecmp) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4501-- REDIR: 0x40da140 (libc.so.6:memrchr) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4501-- REDIR: 0x40ecb20 (libc.so.6:wcslen) redirected to 0x4026560 (
_vgnU_ifunc_wrapper)
--4501-- REDIR: 0x40d17e0 (libc.so.6:__GI_strchr) redirected to 0x402ee00 (
__GI_strchr)
--4501-- REDIR: 0x40ccf00 (libc.so.6:malloc) redirected to 0x402c110 (malloc
)
--4501-- REDIR: 0x40d5280 (libc.so.6:strchrnul) redirected to 0x40330b0 (
strchrnul)
--4501-- REDIR: 0x40d30d0 (libc.so.6:__GI_mempcpy) redirected to 0x40332c0 (
__GI_mempcpy)
--4501-- REDIR: 0x40d15c0 (libc.so.6:__GI_strlen) redirected to 0x402f390 (
__GI_strlen)
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 (no inter)

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 (no inter)

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(-1.010000,-1.010000,0.000000) s(1.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed (no inter)

```

```

Co(-1.010000,-1.010000,0.000000) s(1.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed (no inter)

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
minXYZT(-1.000000,0.000000,0.000000,0.000000)-maxXYZT
(2.000000,1.000000,1.000000,1.000000)

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
minXYZT(-1.000000,0.000000,0.000000,0.000000)-maxXYZT
(1.000000,1.000000,1.000000,1.000000)

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(-1.000000,0.250000,0.000000) s(4.000000,0.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(-1.500000,0.000000,0.000000,0.125000)-maxXYZT
(2.500000,1.000000,1.000000,0.500000)

Co(-1.000000,0.250000,0.000000) s(4.000000,0.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(-0.500000,0.000000,0.000000,0.125000)-maxXYZT
(1.500000,1.000000,1.000000,0.500000)

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.250000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed

```

```

minXYZT(0.000000,-1.500000,0.000000,0.125000)-maxXYZT
(1.000000,2.500000,1.000000,0.500000)

Co(0.250000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(0.000000,-0.500000,0.000000,0.125000)-maxXYZT
(1.000000,1.500000,1.000000,0.500000)

Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.900000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(0.000000,-1.500000,0.000000,0.125000)-maxXYZT
(1.400000,2.500000,1.000000,0.500000)

Co(0.900000,-1.000000,0.000000) s(0.000000,4.000000,0.000000) x
(0.500000,0.000000,0.000000) y(0.000000,0.500000,0.000000) z
(0.000000,0.000000,1.000000)
against
Co(0.000000,0.000000,0.000000) s(0.000000,0.000000,0.000000) x
(1.000000,0.000000,0.000000) y(0.000000,1.000000,0.000000) z
(0.000000,0.000000,1.000000)
Succeed
minXYZT(0.000000,-0.500000,0.000000,0.125000)-maxXYZT
(1.400000,1.500000,1.000000,0.500000)

All unit tests 3DTime have succeed.
--4501-- REDIR: 0x40cd470 (libc.so.6:free) redirected to 0x402d2f0 (free)
==4501==
==4501== HEAP SUMMARY:
==4501==      in use at exit: 0 bytes in 0 blocks
==4501==    total heap usage: 1 allocs, 1 frees, 4,096 bytes allocated
==4501==
==4501== All heap blocks were freed -- no leaks are possible
==4501==
==4501== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4501== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
make[2]: Leaving directory '/home/bayashi/GitHub/FMB/3DTime'
/home/bayashi/GitHub/FMB
make[1]: Leaving directory '/home/bayashi/GitHub/FMB'

```

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.