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

P. Baillehache

November 14, 2019

**Abstract**

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

# Contents

**1 Definition** **3**

**2 Solution** **6**
- 2.1 Fourier-Motzkin elimination method . . . . . . . . . . . . . . . 6
- 2.2 Application of the Fourier-Motzkin method to the intersection problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Algorithm** **7**
- 3.1 2D case . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
- 3.2 3D case . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**4 Implementation** **8**
- 4.1 Frames . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  - 4.1.1 Header . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  - 4.1.2 Body . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
- 4.2 FMB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  - 4.2.1 Header . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  - 4.2.2 Body . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
- 4.3 Example of use . . . . . . . . . . . . . . . . . . . . . . . . 31

**5 Validation** **32**
- 5.1 Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
  - 5.1.1 Unit tests . . . . . . . . . . . . . . . . . . . . . . . . 32
  - 5.1.2 Validation against SAT . . . . . . . . . . . . . . . . . . 49
- 5.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
  - 5.2.1 2D . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
  - 5.2.2 3D . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

**6 Qualification** **53**
- 6.1 Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
- 6.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
  - 6.2.1 2D . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
  - 6.2.2 3D . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

**7 Conclusion** **63**

**8 Annex** **63**
- 8.1 SAT implementation . . . . . . . . . . . . . . . . . . . . . . 63
  - 8.1.1 Header . . . . . . . . . . . . . . . . . . . . . . . . . . 63
  - 8.1.2 Body . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

8.2 Makefile . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

# 1 Definition

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and for a tetrahedron:

$$\mathbb{A} = \left\{ \overrightarrow{X} \in [0.0, 1.0]^D, \sum_{i=0}^{D-1} Xi \leq 1.0 \right\} \tag{8}$$

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

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

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

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

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

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

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

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

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

These can in turn be expressed as systems of linear inequations as follows, given the two shortcuts $\overrightarrow{O_{\mathbb{B}_{\mathbb{A}}}} = C_{\mathbb{A}}^{-1}.(\overrightarrow{O_{\mathbb{B}}} - \overrightarrow{O_{\mathbb{A}}})$ and $C_{\mathbb{B}_{\mathbb{A}}} = C_{\mathbb{A}}^{-1}.C_{\mathbb{B}}$, and the notation $M_{rc}$ for the element at column $c$ and row $r$ of the matrix $M$.

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

$$\left\{ \begin{array}{c} X_0 \leq 1.0 \\ -X_0 \leq 0.0 \\ ... \\ X_{D-1} \leq 1.0 \\ -X_{D-1} \leq 0.0 \\ \sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}0i} X_i \leq 1.0 - O_{\mathbb{B}_{\mathbb{A}}0} \\ -\sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}0i} X_i \leq -O_{\mathbb{B}_{\mathbb{A}}0} \\ ... \\ \sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}(D-1)i} X_i \leq 1.0 - O_{\mathbb{B}_{\mathbb{A}}(D-1)} \\ -\sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}(D-1)i} X_i \leq -O_{\mathbb{B}_{\mathbb{A}}(D-1)} \end{array} \right. \tag{13}$$

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

$$\begin{cases} X_0 \leq 1.0 \\ -X_0 \leq 0.0 \\ ... \\ X_{D-1} \leq 1.0 \\ -X_{D-1} \leq 0.0 \\ \sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}0i} X_i \leq 1.0 - O_{\mathbb{B}_{\mathbb{A}}0} \\ -\sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}0i} X_i \leq -O_{\mathbb{B}_{\mathbb{A}}0} \\ ... \\ \sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}(D-1)i} X_i \leq 1.0 - O_{\mathbb{B}_{\mathbb{A}}(D-1)} \\ -\sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}(D-1)i} X_i \leq -O_{\mathbb{B}_{\mathbb{A}}(D-1)} \\ \sum_{i=0}^{D-1} Xi \leq 1.0 \end{cases} \tag{14}$$

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

$$\begin{cases} X_0 \leq 1.0 \\ -X_0 \leq 0.0 \\ ... \\ X_{D-1} \leq 1.0 \\ -X_{D-1} \leq 0.0 \\ \sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}0i} X_i \leq 1.0 - O_{\mathbb{B}_{\mathbb{A}}0} \\ -\sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}0i} X_i \leq -O_{\mathbb{B}_{\mathbb{A}}0} \\ ... \\ \sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}(D-1)i} X_i \leq 1.0 - O_{\mathbb{B}_{\mathbb{A}}(D-1)} \\ -\sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}(D-1)i} X_i \leq -O_{\mathbb{B}_{\mathbb{A}}(D-1)} \\ \sum_{j=0}^{D-1} \left( \sum_{i=0}^{D-1} \left( C_{\mathbb{B}_{\mathbb{A}}ji} \right) X_i \right) \leq 1.0 - \sum_{i=0}^{D-1} O_{\mathbb{B}_{\mathbb{A}}i} \end{cases} \tag{15}$$

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

$$\begin{cases} X_0 \leq 1.0 \\ -X_0 \leq 0.0 \\ ... \\ X_{D-1} \leq 1.0 \\ -X_{D-1} \leq 0.0 \\ \sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}0i} X_i \leq 1.0 - O_{\mathbb{B}_{\mathbb{A}}0} \\ -\sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}0i} X_i \leq -O_{\mathbb{B}_{\mathbb{A}}0} \\ ... \\ \sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}(D-1)i} X_i \leq 1.0 - O_{\mathbb{B}_{\mathbb{A}}(D-1)} \\ -\sum_{i=0}^{D-1} C_{\mathbb{B}_{\mathbb{A}}(D-1)i} X_i \leq -O_{\mathbb{B}_{\mathbb{A}}(D-1)} \\ \sum_{i=0}^{D-1} Xi \leq 1.0 \\ \sum_{j=0}^{D-1} \left( \sum_{i=0}^{D-1} \left( C_{\mathbb{B}_{\mathbb{A}}ji} \right) X_i \right) \leq 1.0 - \sum_{i=0}^{D-1} O_{\mathbb{B}_{\mathbb{A}}i} \end{cases} \tag{16}$$

# 2 Solution

## 2.1 Fourier-Motzkin elimination method

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

$$\left\{\begin{array}{lllll} a_{11}.x_1 & +a_{12}.x_2 & +\cdots & +a_{1n}.x_n & \leq b_1 \\ a_{21}.x_1 & +a_{22}.x_2 & +\cdots & +a_{2n}.x_n & \leq b_2 \\ & & \vdots & & \\ a_{m1}.x_1 & +a_{m2}.x_2 & +\cdots & +a_{mn}.x_n & \leq b_m \end{array}\right. \tag{17}$$

with

$$\begin{array}{c} i \in 1,2,...,m \\ j \in 1,2,...,n \\ x_i \in \mathbb{R} \\ a_{ij} \in \mathbb{R} \\ b_j \in \mathbb{R} \end{array} \tag{18}$$

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

$$\left\{\begin{array}{llllll} x_1 & +a'_{i2}.x_2 & +\cdots & +a'_{in}.x_n & \leq b'_i & (i \in \mathcal{I}_+) \\ & a_{i2}.x_2 & +\cdots & +a_{in}.x_n & \leq b_i & (i \in \mathcal{I}_0) \\ -x_1 & +a'_{i2}.x_2 & +\cdots & +a'_{in}.x_n & \leq b'_i & (i \in \mathcal{I}_-) \end{array}\right. \tag{19}$$

where

$$\mathcal{I}_+ = \{i : a_{i1} > 0.0\}$$

$$\mathcal{I}_0 = \{i : a_{i1} = 0.0\}$$

$$\mathcal{I}_- = \{i : a_{i1} < 0.0\}$$

$$a'_{ij} = a_{ij}/|a_{i1}|$$

$$b'_i = b_i/|a_{i1}|$$

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

$$\left\{ \begin{array}{cc} \sum_{j=2}^{n}((a'_{kj} + a'_{lj}).x_j) \leq b'_k + b'_l & (k \in \mathcal{I}_+, l \in \mathcal{I}_-) \\ \sum_{j=2}^{n}(a_{ij}.x_j) \leq b_i & i \in \mathcal{I}_0 \end{array} \right. \tag{20}$$

and

$$\max_{l \in \mathcal{I}_-}(\sum_{j=2}^{n}(a'_{lj}.x_j) - b'_l) \leq x_1 \leq \min_{k \in \mathcal{I}_+}(b'_k - \sum_{j=2}^{n}(a'_{kj}.x_j)) \tag{21}$$

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

$$\max_{l \in \mathcal{I}''^{\cdots\prime}_-}(-b''^{\cdots\prime}_l) \leq x_n \leq \min_{k \in \mathcal{I}''^{\cdots\prime}_+}(b''^{\cdots\prime}_k) \tag{22}$$

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

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

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

The solution returned by the system resolution is expressed in the Frame B coordinates system. One can get the coordinates in

# 3 Algorithm

In this section I introduce the algorithm of the solution of the previous section.

### 3.1 2D case

```
algo
```

### 3.2 3D case

```
algo
```

# 4 Implementation

In this section I introduce an implementation of the algorithm 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 -------------

#define FRAME_NB_DIM 3

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

typedef enum {
  FrameCuboid,
  FrameTetrahedron
} FrameType;

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

// Axis aligned bounding box structure
typedef struct {
  double min[FRAME_NB_DIM];
  double max[FRAME_NB_DIM];
} AABB;

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

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

// Print the AABB 'that' on stdout
// Output format is (min[0], min[1], min[2])-(max[0], max[1], max[2])
void AABBPrint(const AABB* const that);

// Print the Frame 'that' on stdout
// Output format is (orig[0], orig[1], orig[2])
// (comp[0][0], comp[0][1], comp[0][2])
// (comp[1][0], comp[1][1], comp[1][2])
// (comp[2][0], comp[2][1], comp[2][2])
void FramePrint(const Frame* const that);

// Create a static Frame structure of FrameType 'type',
// at position 'orig' with components 'comp' ([iComp][iAxis])
Frame FrameCreateStatic(
  const FrameType type,
     const double orig[FRAME_NB_DIM],
     const double comp[FRAME_NB_DIM][FRAME_NB_DIM]);

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

// Export the AABB 'bdgBox' from 'that' 's coordinates system to
// the real coordinates system and update 'bdgBox' with the resulting
// AABB
void FrameExportBdgBox(
  const Frame* const that,
   const AABB* const bdgBox,
         AABB* 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 FrameUpdateInv(Frame* const that);

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

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

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

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

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

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

    }

  }

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

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

    for (int iComp = FRAME_NB_DIM;
         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
  FrameUpdateInv(&that);

  // Return the new Frame
  return that;

}

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

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

  #if FRAME_NB_DIM == 2

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

    tic[0][0] = tc[1][1] / det;
    tic[0][1] = -1.0 * tc[0][1] / det;
    tic[1][0] = -1.0 * tc[1][0] / det;
    tic[1][1] = tc[0][0] / det;

  #elif FRAME_NB_DIM == 3

    // 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;

  #else

    fprintf(stderr,
      "FrameUpdateInv: not implemented for dimension %d\n",
      FRAME_NB_DIM);
    exit(1);

  #endif
```

```
}

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

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

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

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

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

  }

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

    qpo[i] = 0.0;

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

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

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

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

      }
    }
  }
}

// Export the AABB 'bdgBox' from 'that' 's coordinates system to
// the real coordinates system and update 'bdgBox' with the resulting
// AABB
void FrameExportBdgBox(
  const Frame* const that,
   const AABB* const bdgBox,
         AABB* 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)[FRAME_NB_DIM] = that->comp;

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

  bbpma[i] = to[i];

  for (int j = FRAME_NB_DIM;
       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, FRAME_NB_DIM);
for (int iVertex = nbVertices;
     iVertex-- && iVertex;) {

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

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

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

  }

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

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

    w[i] = to[i];

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

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

    }
  }

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

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

        bbpmi[i] = w[i];

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

        bbpma[i] = w[i];

      }
    }
  }

}

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

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

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

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

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

  }
  printf(")");

}

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

    printf("%f", that->orig[i]);
    if (i < FRAME_NB_DIM - 1)
      printf(",");
  }
  for (int j = 0;
       j < FRAME_NB_DIM;
       ++j) {
    printf(") (");
    for (int i = 0;
         i < FRAME_NB_DIM;
         ++i) {

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

```
#ifndef __FMB_H_
#define __FMB_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)
bool FMBTestIntersection(
  const Frame* const that,
  const Frame* const tho,
         AABB* const bdgBox);

#endif
```

#### 4.2.2 Body

```
#include "fmb.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))

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

#define EPSILON 0.0000001 //0.001

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

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// Return false if the system becomes inconsistent during elimination,
// else return true
bool ElimVar(
     const int iVar,
  const double (*M)[FRAME_NB_DIM],
  const double* Y,
     const int nbRows,
     const int nbCols,
        double (*Mp)[FRAME_NB_DIM],
        double* Yp,
    int* const nbRemainRows);

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

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

void PrintMY(
  const double (*M)[FRAME_NB_DIM],
  const double* Y,
     const int nbRows,
     const int nbVar) {
```

```
  for (int iRow = 0; iRow < nbRows; ++iRow) {
    for (int iCol = 0; iCol < nbVar; ++iCol) {
      printf("%f ", M[iRow][iCol]);
    }
    printf("| %f\n", Y[iRow]);
  }
}

void PrintM(
  const double (*M)[FRAME_NB_DIM],
     const int nbRows) {
  for (int iRow = 0; iRow < nbRows; ++iRow) {
    for (int iCol = 0; iCol < FRAME_NB_DIM; ++iCol) {
      printf("%f ", M[iRow][iCol]);
    }
    printf("\n");
  }
}

// Eliminate the 'iVar'-th variable in the system 'M'.X<='Y'
// using the Fourier-Motzkin method and return
// the resulting system in 'Mp' and 'Yp', and the number of rows of
// the resulting system in 'nbRemainRows'
// ('M' arrangement is [iRow][iCol])
// Return true if the system becomes inconsistent during elimination,
// else return false
bool ElimVar(
     const int iVar,
  const double (*M)[FRAME_NB_DIM],
  const double* Y,
     const int nbRows,
     const int nbCols,
        double (*Mp)[FRAME_NB_DIM],
        double* Yp,
    int* const nbRemainRows) {

  // Initialize the number of rows in the result system
  *nbRemainRows = 0;

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

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

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

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

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

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

// Declare a variable to memorize if all the coefficients
// are >= 0.0
bool allPositive = true;

// Declare a variable to memorize if all the coefficients
// are null
bool allNull = true;

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

  if (iCol != iVar) {

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

    // If the coefficient is negative
    if (Mp[*nbRemainRows][jCol] < -1.0 * EPSILON) {

      // Memorize that at least one coefficient is not positive
      allPositive = false;

      // Memorize that at least one coefficient is not null
      allNull = false;

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

    // Else, if the coefficient is positive
    } else if (Mp[*nbRemainRows][jCol] > EPSILON) {

      // Memorize that at least one coefficient is not null
      allNull = false;

    }

    ++jCol;

  }

}

Yp[*nbRemainRows] =
  YIRowDivideByFabsMIRowIVar +
  Y[jRow] / fabs(M[jRow][iVar]);

// If at least one coefficient is not null
if (allNull == false) {

  // If all the coefficients are positive and the right side of
  // the inequality is negative
  if (allPositive == true &&
            Yp[*nbRemainRows] < 0.0) {

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

        }

        // If the right side of the inequality if lower than the sum of
        // negative coefficients in the row
        if (Yp[*nbRemainRows] < sumNegCoeff) {

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

        }

      // Else all coefficients are null, if the right side is null
      // or negative
      } else if (Yp[*nbRemainRows] <= 0.0) {

        // The system is inconsistent
        return true;

      }

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

    }

  }

}

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

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

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

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

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

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

      if (iCol != iVar) {

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

```
          ++jCol;

        }

      }

      Yp[*nbRemainRows] = Y[iRow];

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

    }

  }

  return false;

}

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

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

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

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

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

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

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

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

        // Update the maximum bound
        *max = y;
      }

    // Else, if this row has been reduced to the variable in argument
    // and it has a strictly negative coefficient
    } else if (MjRowiVar < -1.0 * EPSILON) {

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

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

        // Update the minimum bound
        *min = y;

      }

    }

  }

}

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

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

//printf("thoProj:\n");
//FramePrint(&thoProj);
//printf("\n--\n");

  // Declare two variables to memorize the system to be solved M.X <= Y
  // (M arrangement is [iRow][iCol])
  double M[4 * FRAME_NB_DIM + 2][FRAME_NB_DIM];
  double Y[4 * FRAME_NB_DIM + 2];

  // Shortcuts
  double (*thoProjComp)[FRAME_NB_DIM] = thoProj.comp;
  double *thoProjOrig = thoProj.orig;

  // For the first 2 * FRAME_NB_DIM rows of the system
  for (int iStepRow = FRAME_NB_DIM;
       iStepRow--;) {

    // Shortcuts
    int iRow = 2 * iStepRow;
    int iNextRow = iRow + 1;
    double* MIRow = M[iRow];
    double* MINextRow = M[iNextRow];
    int jRow = 2 * (iStepRow + FRAME_NB_DIM);
    int jNextRow = jRow + 1;
    double* MJRow = M[jRow];
    double* MJNextRow = M[jNextRow];

    // Declare a variable to memorize the sum of the negative
    // coefficients in the row
    double sumNegCoeff[2] = {0.0, 0.0};

    // Declare a variable to memorize if all the coefficients
    // are >= 0.0
    bool allPositive[2] = {true, true};

    // Declare a variable to memorize if all the coefficients
    // are null
    bool allNull = true;

    // For each column of the system
    for (int iCol = FRAME_NB_DIM;
         iCol--;) {

      MIRow[iCol] = thoProjComp[iCol][iStepRow];
      MINextRow[iCol] = -1.0 * MIRow[iCol];

      // If it's on the diagonal
      if (iStepRow == iCol) {

        MJRow[iCol] = 1.0;
        MJNextRow[iCol] = -1.0;

      // Else it's not on the diagonal
      } else {

        MJRow[iCol] = 0.0;
        MJNextRow[iCol] = 0.0;

      }

      // If the coefficient is negative
      if (MIRow[iCol] < -1.0 * EPSILON) {

        // Memorize that at least one coefficient is not positive
        allPositive[0] = false;

        // Memorize that at least one coefficient is not null
        allNull = false;

        // Update the sum of the negative coefficient
        sumNegCoeff[0] += MIRow[iCol];

      // Else, if the coefficient is positive
      } else if (MIRow[iCol] > EPSILON) {

        // Memorize that at least one coefficient is not null
        allNull = false;

        // Memorize that at least one coefficient is not positive
        allPositive[1] = false;
```

```
      // Update the sum of the negative coefficient
      sumNegCoeff[1] -= MIRow[iCol];

    }

  }

  Y[iRow] = 1.0 - thoProjOrig[iStepRow];
  Y[iNextRow] = thoProjOrig[iStepRow];

  // If at least one coefficient is not null
  if (allNull == false) {

    // If all the coefficients are positive and the right side of the
    // inequality is negative
    if ((allPositive[0] == true &&
         Y[iRow] < 0.0) ||
        (allPositive[1] == true &&
         Y[iNextRow] < 0.0)) {

      // As X is in [0,1], the system is inconsistent,
      // there is no intersection
      return false;

    }

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[iRow] < sumNegCoeff[0] ||
        Y[iNextRow] < sumNegCoeff[1]) {

      // As X is in [0,1], the system is inconsistent
      // there is no intersection
      return false;

    }

  // Else all coefficients are null, if the right side is null
  // or negative
  } else if (Y[iRow] <= 0.0 ||
             Y[iNextRow] <= 0.0) {

    // The system is inconsistent, there is no intersection
    return false;

  }

  Y[jRow] = 1.0;
  Y[jNextRow] = 0.0;

}

// Declare a variable to memorize the total number of rows in the
// system. It may vary depending on the type of Frames
int nbRows = 4 * FRAME_NB_DIM;

// If the first frame is a Tetrahedron
if (that->type == FrameTetrahedron) {

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

```
// Declare a variable to memorize if all the coefficients
// are >= 0.0
bool allPositive = true;

// Declare a variable to memorize if all the coefficients
// are null
bool allNull = true;

// Shortcut
double* MRow = M[nbRows];

Y[nbRows] = 1.0;

// For each column of the system
for (int iCol = FRAME_NB_DIM;
     iCol--;) {

  MRow[iCol] = 0.0;

  // For each component
  for (int iAxis = FRAME_NB_DIM;
       iAxis--;) {

    MRow[iCol] += thoProjComp[iCol][iAxis];

  }

  Y[nbRows] -= thoProjOrig[iCol];

  // If the coefficient is negative
  if (MRow[iCol] < -1.0 * EPSILON) {

    // Memorize that at least one coefficient is not positive
    allPositive = false;

    // Memorize that at least one coefficient is not null
    allNull = false;

    // Update the sum of the negative coefficient
    sumNegCoeff += MRow[iCol];

  // Else, if the coefficient is positive
  } else if (MRow[iCol] > EPSILON) {

    // Memorize that at least one coefficient is not null
    allNull = false;

  }

}

// If at least one coefficient is not null
if (allNull == false) {

  // If all the coefficients are positive and the right side of the
  // inequality is negative
  if (allPositive == true &&
       Y[nbRows] < 0.0) {

    // As X is in [0,1], the system is inconsistent,
    // there is no intersection
```

```
      return false;

    }

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[nbRows] < sumNegCoeff) {

      // As X is in [0,1], the system is inconsistent
      // there is no intersection
      return false;

    }

  // Else all coefficients are null, if the right side is null
  // or negative
  } else if (Y[nbRows] <= 0.0) {

    // The system is inconsistent, there is no intersection
    return false;

  }

  // Update the number of rows in the system
  ++nbRows;

}

// If the second frame is a Tetrahedron
if (tho->type == FrameTetrahedron) {

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

  // Declare a variable to memorize if all the coefficients
  // are >= 0.0
  bool allPositive = true;

  // Declare a variable to memorize if all the coefficients
  // are null
  bool allNull = true;

  // Shortcut
  double* MRow = M[nbRows];

  // For each column of the system
  for (int iCol = FRAME_NB_DIM;
       iCol--;) {

    MRow[iCol] =  1.0;

    // If the coefficient is negative
    if (MRow[iCol] < -1.0 * EPSILON) {

      // Memorize that at least one coefficient is not positive
      allPositive = false;

      // Memorize that at least one coefficient is not null
      allNull = false;

      // Update the sum of the negative coefficient
      sumNegCoeff += MRow[iCol];

    // Else, if the coefficient is positive
    } else if (MRow[iCol] > EPSILON) {

      // Memorize that at least one coefficient is not null
      allNull = false;

    }

  }

  Y[nbRows] = 1.0;

  // If at least one coefficient is not null
  if (allNull == false) {

    // If all the coefficients are positive and the right side of the
    // inequality is negative
    if (allPositive == true &&
        Y[nbRows] < 0.0) {

      // As X is in [0,1], the system is inconsistent,
      // there is no intersection
      return false;

    }

    // If the right side of the inequality is lower than the sum of
    // negative coefficients in the row
    if (Y[nbRows] < sumNegCoeff) {

      // As X is in [0,1], the system is inconsistent
      // there is no intersection
      return false;

    }

  // Else all coefficients are null, if the right side is null
  // or negative
  } else if (Y[nbRows] <= 0.0) {

    // The system is inconsistent, there is no intersection
    return false;

  }

  // Update the number of rows in the system
  ++nbRows;

}

// Solve the system

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

// Declare variables to eliminate the first variable
// The number of rows is set conservatively, one may try to reduce
// them if needed
double Mp[4 * FRAME_NB_DIM + 2 + 50][FRAME_NB_DIM];
double Yp[4 * FRAME_NB_DIM + 2 + 50];
int nbRowsP;

// Eliminate the first variable
bool inconsistency =
  ElimVar(
    FST_VAR,
    M,
    Y,
    nbRows,
    FRAME_NB_DIM,
    Mp,
    Yp,
    &nbRowsP);

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

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

}

#if FRAME_NB_DIM == 3

  // Declare variables to eliminate the second variable
  // The number of rows is set conservatively, one may try to reduce
  // them if needed
  double Mpp[4 * FRAME_NB_DIM + 2 + 500][FRAME_NB_DIM];
  double Ypp[4 * FRAME_NB_DIM + 2 + 500];
  int nbRowsPP;

  // Eliminate the second variable (which is the first in the new system)
  inconsistency =
    ElimVar(
      FST_VAR,
      Mp,
      Yp,
      nbRowsP,
      FRAME_NB_DIM - 1,
      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
  GetBound(
    THD_VAR,
    Mpp,
    Ypp,
    nbRowsPP,
    &bdgBoxLocal);

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

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

}

// Eliminate the third variable (which is second first in the new
// system)
inconsistency =
  ElimVar(
    SND_VAR,
    Mp,
    Yp,
    nbRowsP,
    FRAME_NB_DIM - 1,
    Mpp,
    Ypp,
    &nbRowsPP);

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

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

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

  // Immediately return true
  return true;

}

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

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

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

}

// Now starts again from the initial systems and eliminate the
// second and third variables to get the bounds of the first variable
inconsistency =
  ElimVar(
    SND_VAR,
    M,
    Y,
    nbRows,
    FRAME_NB_DIM,
    Mp,
    Yp,
        &nbRowsP);

    if (inconsistency == true) {
      return false;
    }

    inconsistency =
      ElimVar(
        SND_VAR,
        Mp,
        Yp,
        nbRowsP,
        FRAME_NB_DIM - 1,
        Mpp,
        Ypp,
        &nbRowsPP);

    if (inconsistency == true) {
      return false;
    }

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

    if (bdgBoxLocal.min[FST_VAR] >= bdgBoxLocal.max[FST_VAR]) {
      return false;
    }

  #elif FRAME_NB_DIM == 2

//PrintMY(M, Y, nbRows, 2);
//printf("--\n");
//PrintMY(Mp, Yp, nbRowsP, 1);

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

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

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

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

      // Immediately return true
      return true;

    }
    // Now starts again from the initial systems and eliminate the
    // second variable to get the bounds of the first variable
    inconsistency =
      ElimVar(
        SND_VAR,
        M,
        Y,
        nbRows,
        FRAME_NB_DIM,
        Mp,
        Yp,
        &nbRowsP);
    if (inconsistency == true) {
      return false;
    }

//printf("--\n");
//PrintMY(Mp, Yp, nbRowsP, 1);

    GetBound(
      FST_VAR,
      Mp,
      Yp,
      nbRowsP,
      &bdgBoxLocal);

//printf("--\nbdgBoxLocal:\n");
//AABBPrint(&bdgBoxLocal);printf("\n");

    if (bdgBoxLocal.min[FST_VAR] >= bdgBoxLocal.max[FST_VAR]) {
      return false;
    }

  #else

    printf("Not implemented for dimension %d\n", FRAME_NB_DIM);
    exit(0);

  #endif

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

    // Export the local bounding box toward the real coordinates
    // system
    FrameExportBdgBox(tho, &bdgBoxLocal, bdgBox);

//printf("bdgBox:\n");AABBPrint(bdgBox);printf("\n");
//printf("that->bdgBox:\n");AABBPrint(&(that->bdgBox));printf("\n");

    // Clip with the AABB of 'that'
    double* const min = bdgBox->min;
    double* const max = bdgBox->max;
    const double* const thatBdgBoxMin = that->bdgBox.min;
    const double* const thatBdgBoxMax = that->bdgBox.max;
    for (int iAxis = FRAME_NB_DIM;
         iAxis--;) {

      if (min[iAxis] < thatBdgBoxMin[iAxis]) {

        min[iAxis] = thatBdgBoxMin[iAxis];
      }
      if (max[iAxis] > thatBdgBoxMax[iAxis]) {

        max[iAxis] = thatBdgBoxMax[iAxis];

      }

    }

  }

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

}
```

## 4.3 Example of use

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

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

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

  // Create the two objects to be tested for intersection
#if FRAME_NB_DIM == 3
  double origP[FRAME_NB_DIM] = {0.0, 0.0, 0.0};
  double compP[FRAME_NB_DIM][FRAME_NB_DIM] = {
    {1.0, 0.0, 0.0},  // First component
    {0.0, 1.0, 0.0},  // Second component
    {0.0, 0.0, 1.0}}; // Third component
#elif FRAME_NB_DIM == 2
  double origP[FRAME_NB_DIM] = {0.0, 0.0};
  double compP[FRAME_NB_DIM][FRAME_NB_DIM] = {
    {1.0, 0.0},  // First component
    {0.0, 1.0}}; // Second component
#else
  printf("Not implemented for dimension %d\n", FRAME_NB_DIM);
  exit(0);
#endif
  Frame P =
    FrameCreateStatic(
      FrameTetrahedron,
      origP,
      compP);

#if FRAME_NB_DIM == 3
  double origQ[FRAME_NB_DIM] = {0.5, 0.5, 0.5};
  double compQ[FRAME_NB_DIM][FRAME_NB_DIM] = {
    {2.0, 0.0, 0.0},
    {0.0, 2.0, 0.0},
    {0.0, 0.0, 2.0}};
#elif FRAME_NB_DIM == 2
  double origQ[FRAME_NB_DIM] = {0.5, 0.5};
  double compQ[FRAME_NB_DIM][FRAME_NB_DIM] = {
    {-1.0, 0.0},
    {0.0, -1.0}};
#else
  printf("Not implemented for dimension %d\n", FRAME_NB_DIM);
  exit(0);
#endif
  Frame Q =
    FrameCreateStatic(
      FrameTetrahedron,
      origQ,
      compQ);

  // Declare a variable to memorize the result of the intersection
  // detection
  AABB bdgBox;

  // Test for intersection between P and Q
  bool isIntersecting = FMBTestIntersection(&P, &Q, &bdgBox);

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

    printf("Intersection detected in AABB ");
    AABBPrint(&bdgBox);
    printf("\n");

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

    printf("No intersection.\n");

  }

  return 0;
}
```

# 5 Validation

In this section I introduce the code I've used to validate the algorithm and its implementation. The validation consists of, first running the algorithm on a set of unit test for which the solution has been computed by hand, and second 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.63)

## 5.1 Code

### 5.1.1 Unit tests

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

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

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

// Epslon for numerical precision
#define EPSILON 0.0001

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

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

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

  Frame Q =
    FrameCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.comp);

  // Declare a variable to memorize the resulting bounding box
  AABB bdgBox;

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

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

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

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

    // 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) {

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

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

            flag = false;

          }

        }

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

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

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

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

          // Terminate the unit tests
          exit(0);

        }

      // Else the Frames were not intersected,
      // no need to check the bounding box
```

```
      } else {

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

      }

    }
    printf("\n");

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

  }

}

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

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

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

  // Execute the unit test on various cases

#if FRAME_NB_DIM == 3

  // ---------------------------
  paramP = (Param)
    {.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 = (Param)
    {.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 = (AABB)
    {.min = {0.0, 0.0, 0.0},
     .max = {1.0, 1.0, 1.0}
    };
  UnitTest(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

  // ---------------------------
  paramP = (Param)
  {.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 = (Param)
  {.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 = (AABB)
  {.min = {0.5, 0.5, 0.5},
   .max = {1.0, 1.0, 1.0}
  };
UnitTest(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param)
  {.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 = (Param)
  {.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 = (AABB)
  {.min = {0.0, 0.0, 0.0},
   .max = {0.5, 0.5, 0.5}
  };
UnitTest(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param)
  {.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 = (Param)
```

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

// ----------------------------
paramP = (Param)
  {.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 = (Param)
  {.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}}
  };
UnitTest(
  paramP,
  paramQ,
  false,
  NULL);

// ----------------------------
paramP = (Param)
  {.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 = (Param)
  {.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 = (AABB)
  {.min = {0.5, 0.5, -1.0},
   .max = {1.0, 1.0, -0.5}
  };
UnitTest(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param)
  {.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 = (Param)
  {.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}}
  };
UnitTest(
  paramP,
  paramQ,
  false,
  NULL);

// ----------------------------
paramP = (Param)
  {.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 = (Param)
  {.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 = (AABB)
  {.min = {0.0, -0.5, 0.0},
   .max = {1.0, 0.0, 1.0}
  };
UnitTest(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param)
  {.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 = (Param)
  {.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}}
  };
UnitTest(
  paramP,
  paramQ,
  false,
  NULL);

// ----------------------------
paramP = (Param)
  {.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 = (Param)
  {.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 = (AABB)
  {.min = {0.0, -0.5, 0.0},
   .max = {0.75, 0.0, 0.75}
  };
UnitTest(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

// ----------------------------
paramP = (Param)
  {.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 = (Param)
  {.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}}
  };
UnitTest(
  paramP,
    paramQ,
    false,
    NULL);

  // ---------------------------
  paramP = (Param)
    {.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 = (Param)
    {.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 = (AABB)
    {.min = {0.0, -0.5, 0.0},
     .max = {0.5, -0.5 + 1.0 / 3.0, 0.5}
    };
  UnitTest(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

#elif FRAME_NB_DIM == 2

  // ---------------------------
  paramP = (Param)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0},
     .comp =
       {{1.0, 0.0},
        {0.0, 1.0}}
    };
  paramQ = (Param)
    {.type = FrameCuboid,
     .orig = {0.0, 0.0},
     .comp =
       {{1.0, 0.0},
        {0.0, 1.0}}
    };
  correctBdgBox = (AABB)
    {.min = {0.0, 0.0},
     .max = {1.0, 1.0}
    };
  UnitTest(
    paramP,
    paramQ,
    true,
    &correctBdgBox);

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

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

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

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

// ----------------------------
paramP = (Param)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 1.0},
      {-1.0, 1.0}}
  };
paramQ = (Param)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
correctBdgBox = (AABB)
  {.min = {0.0, 0.0},
   .max = {1.0, 1.0}
  };
UnitTest(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

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

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

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

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

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

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

// ---------------------------
paramP = (Param)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.5},
      {0.5, 1.0}}
  };
paramQ = (Param)
  {.type = FrameCuboid,
   .orig = {1.0, 2.0},
   .comp =
     {{-0.5, -0.5},
      {0.0, -1.0}}
  };
correctBdgBox = (AABB)
  {.min = {0.5, 0.75},
   .max = {1.0, 1.25}
  };
UnitTest(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

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

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

// ---------------------------
paramP = (Param)
  {.type = FrameCuboid,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
paramQ = (Param)
  {.type = FrameTetrahedron,
   .orig = {0.0, 0.0},
   .comp =
     {{1.0, 0.0},
      {0.0, 1.0}}
  };
correctBdgBox = (AABB)
  {.min = {0.0, 0.0},
   .max = {1.0, 1.0}
  };
UnitTest(
  paramP,
  paramQ,
  true,
  &correctBdgBox);

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

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

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

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

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

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

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

#else

  printf("Not implemented for dimension %d\n", FRAME_NB_DIM);
  exit(0);

#endif

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

  return 0;
}
```

### 5.1.2 Validation against SAT

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

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

// Espilon for numerical precision
#define EPSILON 0.0001
// 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[FRAME_NB_DIM];
  double comp[FRAME_NB_DIM][FRAME_NB_DIM];
} Param;

// 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 Validation(
  const Param paramP,
  const Param paramQ) {

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

  Frame Q =
    FrameCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.comp);

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

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

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

    // Test intersection with SAT
#if FRAME_NB_DIM == 2

    bool isIntersectingSAT =
      SATTestIntersection2D(
        that,
        tho);

#elif FRAME_NB_DIM == 3

    bool isIntersectingSAT =
      SATTestIntersection3D(
        that,
        tho);

#else

    printf("Not implemented for dimension %d\n", FRAME_NB_DIM);
    exit(0);

#endif

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

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

      // Stop the validation
      exit(0);

    }

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

      // Update the number of intersection
      nbInter++;

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

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

    }

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

  }

}

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

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

  // Declare two variables to memozie the arguments to the
  // Validation function
  Param paramP;
  Param 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
    Param* 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 = FRAME_NB_DIM;
           iAxis--;) {

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

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

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

        }

      }

      param = &paramQ;

    }

    // Calculate the determinant of the Frames' components matrix
#if FRAME_NB_DIM == 2

    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];

#elif FRAME_NB_DIM == 3

    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]);

#else

  printf("Not implemented for dimension %d\n", FRAME_NB_DIM);
  exit(0);

#endif

    // 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
      Validation(
        paramP,
        paramQ);

    }

  }

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

  return 0;
}
```

## 5.2 Results

### 5.2.1 2D

### 5.2.2 3D

# 6 Qualification

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.

## 6.1 Code

```
// 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 "fmb.h"
#include "sat.h"

// Espilon for numerical precision
#define EPSILON 0.0001
// Range of values for the random generation of Frames
#define RANGE_AXIS 100.0
// Nb of run
#define NB_RUNS 1
// Nb of tests per run
#define NB_TESTS 1000000
// Nb of times the test is run on one pair of frame, used to
// slow down the processus and be able to measure time
```

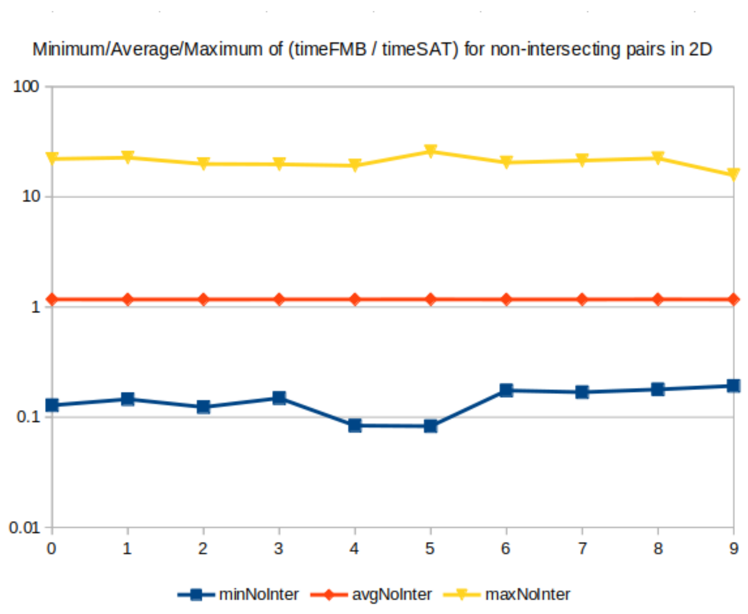
```
#if FRAME_NB_DIM == 3
  #define NB_REPEAT 800
#elif FRAME_NB_DIM == 2
  #define NB_REPEAT 1500
#endif

// 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[FRAME_NB_DIM];
  double comp[FRAME_NB_DIM][FRAME_NB_DIM];
} Param;

// 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;

// 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 Qualification(
        const Param paramP,
        const Param paramQ) {

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

  Frame Q =
    FrameCreateStatic(
      paramQ.type,
      paramQ.orig,
      paramQ.comp);

  // Helper variables to loop on the pair (that, tho) and (tho, that)
  Frame* that = &P;
  Frame* 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] = {false};

    // Start measuring time
    struct timeval start;
    gettimeofday(&start, NULL);

    // Run the FMB intersection test
    for (int i = NB_REPEAT;
         i--;) {

      isIntersectingFMB[i] =
        FMBTestIntersection(
          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] = {false};

    // Start measuring time
    gettimeofday(&start, NULL);

    // Run the FMB intersection test
    for (int i = NB_REPEAT;
         i--;) {

#if FRAME_NB_DIM == 2

      isIntersectingSAT[i] =
        SATTestIntersection2D(
          that,
          tho);

#elif FRAME_NB_DIM == 3

      isIntersectingSAT[i] =
        SATTestIntersection3D(
          that,
          tho);

#else
```

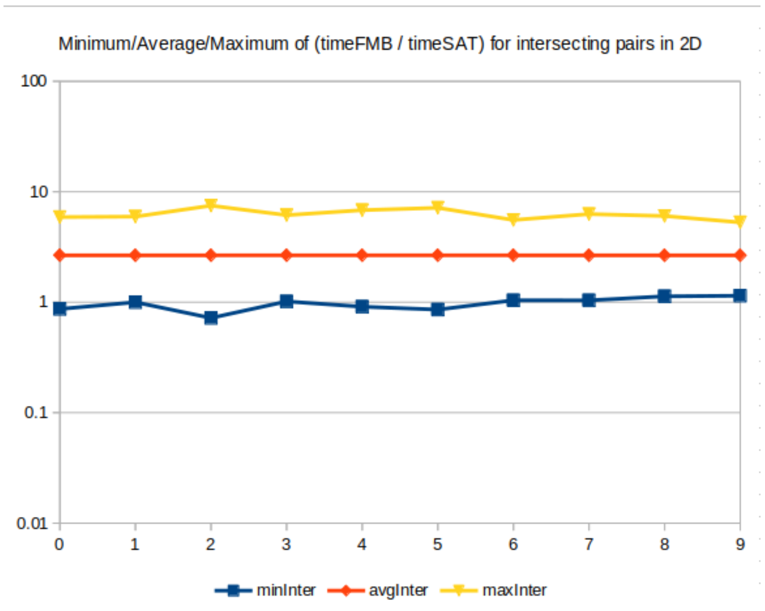
```
      printf("Not implemented for dimension %d\n", FRAME_NB_DIM);
      exit(0);

#endif
    }

    // 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");
        FramePrint(that);
        printf(" against ");
        FramePrint(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 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;

   // 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;

   }

  // 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;

 }

}

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

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

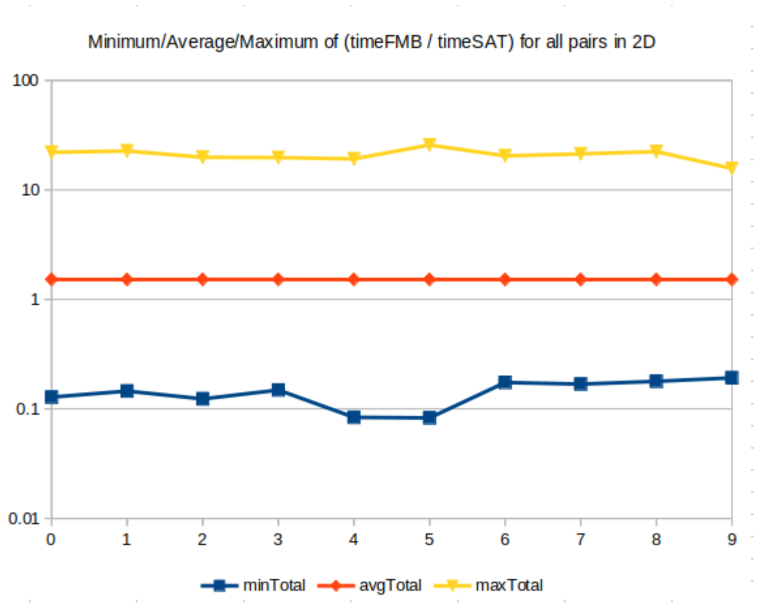
```
// Loop on runs
for (int iRun = 0;
     iRun < NB_RUNS;
     ++iRun) {

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

  // Declare two variables to memozie the arguments to the
  // Qualification function
  Param paramP;
  Param paramQ;

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

    // Create two random Frame definitions
    Param* 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 = FRAME_NB_DIM;
           iAxis--;) {

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

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

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

        }

      }

      param = &paramQ;

    }

    // Calculate the determinant of the Frames' components matrix
#if FRAME_NB_DIM == 2

    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];

#elif FRAME_NB_DIM == 3

    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]);

#else

  printf("Not implemented for dimension %d\n", FRAME_NB_DIM);
  exit(0);

#endif

    // 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
      Qualification(
        paramP,
        paramQ);

    }

  }

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

    printf("ratio (timeFMB / timeSAT)\n");
    printf("run\tcountInter\tcountNoInter\t");
    printf("minInter\tavgInter\tmaxInter\t");
    printf("minNoInter\tavgNoInter\tmaxNoInter\t");
    printf("minTotal\tavgTotal\tmaxTotal\n");

  }

  printf("%d\t%lu\t%lu\t", iRun, countInter, countNoInter);

  double avgInter = sumInter / (double)countInter;
  printf("%f\t%f\t%f\t", minInter, avgInter, maxInter);

  double avgNoInter = sumNoInter / (double)countNoInter;
  printf("%f\t%f\t%f\t", minNoInter, avgNoInter, maxNoInter);

  double avg =
    (sumInter + sumNoInter) / (double)(countInter + countNoInter);
  printf("%f\t%f\t%f\n",
      (minNoInter < minInter ? minNoInter : minInter),
      avg,
      (maxNoInter > maxInter ? maxNoInter : maxInter));

  }

  return 0;

}
```

## 6.2 Results

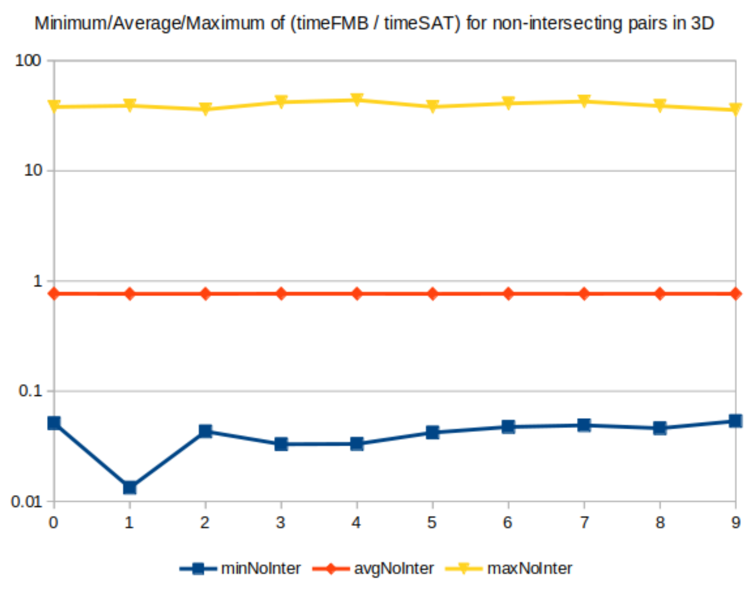
### 6.2.1 2D

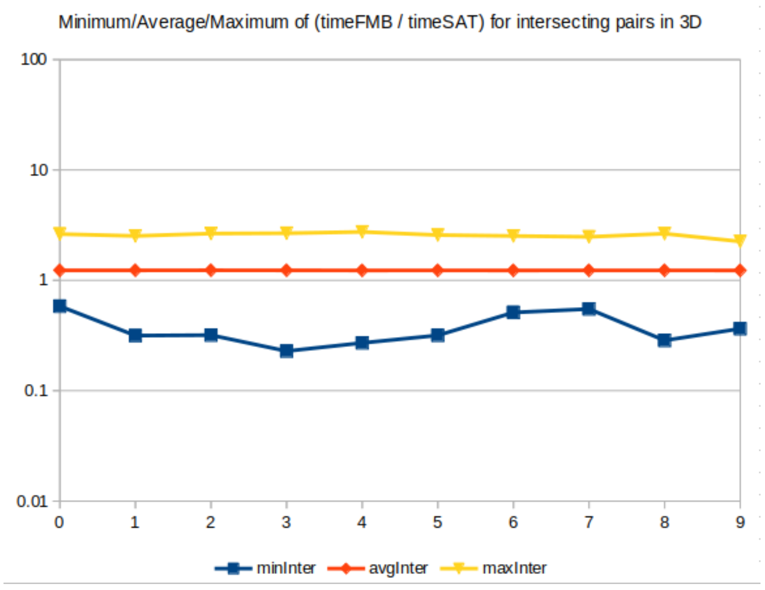


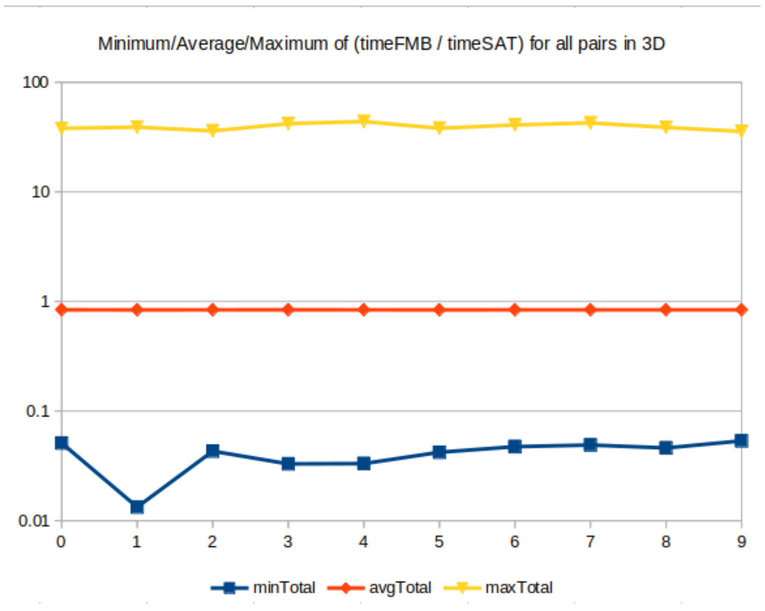

Minimum/Average/Maximum of (timeFMB / timeSAT) for intersecting pairs in 2D
100
10
1
0.1
0.01
0
1
2
3
4
5
6
7
8
9
minInter
avgInter
maxInter


Minimum/Average/Maximum of (timeFMB / timeSAT) for all pairs in 2D
100
10
1
0.1
0.01
0
1
2
3
4
5
6
7
8
9
minTotal
avgTotal
maxTotal

### 6.2.2 3D


Minimum/Average/Maximum of (timeFMB / timeSAT) for non-intersecting pairs in 3D
100
10
1
0.1
0.01
0
1
2
3
4
5
6
7
8
9
minNoInter
avgNoInter
maxNoInter


Minimum/Average/Maximum of (timeFMB / timeSAT) for intersecting pairs in 3D
100
10
1
0.1
0.01
0
1
2
3
4
5
6
7
8
9
minInter
avgInter
maxInter



## 7 Conclusion

The validation proves that the FMB algorithm correctly identifies intersection of pairs of Frames in accordance with the results of the SAT algorithm.

The qualification proves that the FMB algorithm is in average 50% slower than the SAT algorithm in 2D, and 17% faster in 3D.

## 8 Annex

### 8.1 SAT implementation

In this section I introduce the code of the implementation of the SAT algorithm, used to validate and qualify the FMB algorithm.

#### 8.1.1 Header

```
#ifndef __SAT_H_
#define __SAT_H_

#include <stdbool.h>
```

```
#include <string.h>
#include "frame.h"

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

#if FRAME_NB_DIM == 2
// Test for intersection between 2D Frame 'that' and 2D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2D(
  const Frame* const that,
  const Frame* const tho);
#endif

#if FRAME_NB_DIM == 3
// Test for intersection between 3D Frame 'that' and 3D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3D(
  const Frame* const that,
  const Frame* const tho);
#endif

#endif
```

### 8.1.2 Body

```
#include "sat.h"

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

#define EPSILON 0.0000001

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

#if FRAME_NB_DIM == 3
// Check the intersection constraint along one axis
bool CheckAxis(
  const Frame* const that,
  const Frame* const tho,
  const double* const axis);
#endif

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

#if FRAME_NB_DIM == 2
// Test for intersection between 2D Frame 'that' and 2D Frame 'tho'
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection2D(
  const Frame* const that,
  const Frame* const tho) {

  // Declare a variable to loop on Frames and commonalize code
  const Frame* frameEdge = that;

  // Loop to commonalize code when checking SAT based on that's edges
  // and then tho's edges
  for (int iTest = 2;
       iTest--;) {

    // 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 Frame* 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;

}
#endif

#if FRAME_NB_DIM == 3
// Test for intersection between 3D Frame ’that’ and 3D Frame ’tho’
// Return true if the two Frames are intersecting, else false
bool SATTestIntersection3D(
  const Frame* const that,
  const Frame* 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 Frame* 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 iPair = 2;
     iPair--;) {

  // 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 =
    CheckAxis(
      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 =
        CheckAxis(
          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;

}

// Check the intersection constraint for Frames 'that' and 'tho'
// relatively to 'axis'
bool CheckAxis(
  const Frame* const that,
  const Frame* 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 Frame* 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;

}

#endif
```

## 8.2 Makefile

In this section I introduce the Makefile used to compile the code given in the previous sections.

```
all : main unitTests validation qualification

BUILD_ARG=-O3

main : main.o fmb.o sat.o frame.o Makefile
gcc -o main main.o fmb.o sat.o frame.o

main.o : main.c fmb.h frame.h Makefile
gcc -c main.c $(BUILD_ARG)

unitTests : unitTests.o fmb.o frame.o Makefile
gcc -o unitTests unitTests.o fmb.o frame.o $(LINK_ARG)

unitTests.o : unitTests.c fmb.h frame.h Makefile
gcc -c unitTests.c $(BUILD_ARG)

validation : validation.o fmb.o sat.o frame.o Makefile
gcc -o validation validation.o fmb.o sat.o frame.o $(LINK_ARG)

validation.o : validation.c fmb.h sat.h frame.h Makefile
gcc -c validation.c $(BUILD_ARG)

qualification : qualification.o fmb.o sat.o frame.o Makefile
gcc -o qualification qualification.o fmb.o sat.o frame.o $(LINK_ARG)

qualification.o : qualification.c fmb.h sat.h frame.h Makefile
gcc -c qualification.c $(BUILD_ARG)

fmb.o : fmb.c fmb.h frame.h Makefile
gcc -c fmb.c $(BUILD_ARG)

sat.o : sat.c sat.h frame.h Makefile
gcc -c sat.c $(BUILD_ARG)

frame.o : frame.c frame.h Makefile
gcc -c frame.c $(BUILD_ARG)

clean :
rm -f *.o main unitTests validation qualification

valgrind :
valgrind -v --track-origins=yes --leak-check=full \
--gen-suppressions=yes --show-leak-kinds=all ./main
```

## References

[1] J.J.-B. Fourier. Oeuvres II. Paris, 1890

[2] T.S. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen.* Thesis, 1936. Reprinted in: *Theodore S. Motzkin: selected papers* (D.Cantor et al., eds,), Birkhäuser, Boston, 1983.

[3] Dynamic Collision Detection using Oriented Bounding Boxes, David Eberly, Geometric Tools, Redmond WA 98052