Introduction to Numerical Computing: Numerically Stable Collision Detection

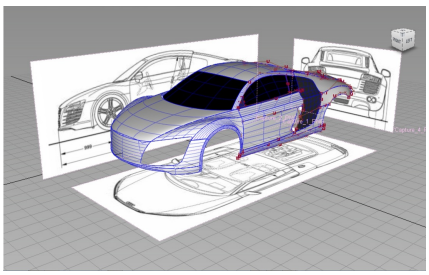
Abstract

This tutorial will provide a quick introduction to the themes of numerical calculation. After analysing how real numbers are represented in a discrete machine and analysing several possible error scenarios, we will discuss three different approaches to making the numerically stable algorithms. Either handle errors through confidence intervals, or make numerically stable specific operations that are the hare of the most error numbers in the algorithm, or entirely rewrite such algorithm in a way that avoids such operation and overall minimises the amount of numeric operations.

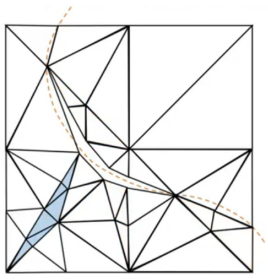
1 Introduction

Numerical computing is used in all those contexts where it is required to process a considerable amount of data for automating and fastening decision-making and analysis processes. To realize this mechanism, we must undergo some preliminary modelling steps:

1. **Real World to Mathematical Model**. In all scientific disciplines, a mathematical model is introduced to describe a real object or phenomenon via a necessary approximation (*model*) to limit the number of factors to be considered unmanageable otherwise. Some examples:
   - In CAD softwares (Figure 1a), the curves and surfaces are only an approximation of those of real objects. In polygonal modelers, the continuous surfaces representing the body of a car [8], are approximated by a series of small flat faces (e.g., triangles) joined together (*mesh*).
   - After modelling an object as a mesh, we can further approximate its physical properties and simulate cutting operations (Figure 1b): this technique has a medical applications to provide realistic simulations of surgical operations.
2. **Mathematical Model to Computational Representation**. Theoretically described mathematical models cannot be always represented and solved precisely over a discrete machine (*computer*). The purpose of Numerical Computing is to study the effects of the approximations introduced in these steps.



(a) Polygonal model and orthographic projections of a car in a CAD software.



(b) Represting a cut with a *Bezier Curve* and then subdividing the surface via triangulation generation [11].

Figure 1: Examples of mathematical models that are used as intermediate steps for achieving computational representation.

The need to be able to approximate continuous mathematical problems (e.g., in $\mathbb{R}^3$) in a discrete representation arises from the architectural limits of modern electronic computers: although these can perform both *symbolic* (and therefore with exact accuracy, e.g. SageMath [1] and [1]) and *numerical computations* in a finite representation (**floating-point arithmetic**), modern machines are structured in such a way that numerical calculations can be performed far more efficiently than symbolic computations, albeit with some error. Therefore, in this tutorial we're going to focus on numerical computations.

Because computers have necessarily a limited amount of memory, they will never be able to represent irrational numbers with infinite precision in floating-point arithmetic, where each number can be only represented by a fixed number of bits: it follows that, inevitably, we can represent only a fixed amount of real numbers, and that all computations will be affected by errors.

1.1 Representing Real Numbers ($\mathbb{R}$)

At the very root of the impossibility of accurately implementing mathematical models based on a continuous set of symbols is the numeric representation of real numbers used in computers. For example, numbers such as π or $\sqrt{2}$, whose representation requires an infinite number of digits, can only be represented approximately, thus committing a representation error. Also, given that the set of representable numbers is finite and therefore limited, arbitrarily small or arbitrarily large numbers cannot be represented.

In order to understand this, we have first to know how to convert the numbers from decimal into binary representation.

Definition 1 (Basis Representation)**.** *The representation of a real number $\alpha \neq 0$ under a **base** $\beta \geq 2$, consists of an unique **exponent** $p \in \mathbb{N}$ and some integers $\{\alpha_i\}_{i \in \mathbb{N} \setminus \{0\}}$ such that $\alpha_1 \neq 0$ and $0 \leq \alpha_i < \beta$ such that:*

$$\alpha = \pm \left(\sum_{i=0}^{\infty} \alpha_i \beta^{-i} \right) \beta^p$$

*where $m = \sum_{i=0}^{\infty} \alpha_i \beta^{-i}$ is called **mantissa**.*

Example 1. *Let us suppose that we want to represent 89 in decimal representation (89_{10}) into a binary number ($\beta = 2$), e.g., stored in an `unsigned int`. The algorithm that we have to follow is the following: divide 89 by 2 up until we obtain a quotient which is smaller than β. Therefore, we will obtain the following result:*

$$\begin{aligned}
\frac{89}{2} &= 44 + \frac{\boldsymbol{1}}{2} \\
\frac{44}{2} &= 22 + \frac{\boldsymbol{0}}{2} \\
\frac{22}{2} &= 11 + \frac{\boldsymbol{0}}{2} \\
\frac{11}{2} &= 5 + \frac{\boldsymbol{1}}{2} \\
\frac{5}{2} &= 2 + \frac{\boldsymbol{1}}{2} \\
\frac{2}{2} &= \boldsymbol{1} + \frac{\boldsymbol{0}}{2}
\end{aligned}$$

If we read the numbers in bold, then we will obtain 1011001_2, which is the binary representation of 89_{10}. Please observe that the number of digits of 89_{10} in binary representation corresponds to $\lceil \log_2(89_{10}) \rceil$. By taking the basis representation definition, we can also transform the binary number in decimal representation: in fact, if we choose $p = \lceil \log_2(89_{10}) \rceil$, we have: $\boldsymbol{1} \cdot 2^6 + \boldsymbol{1} \cdot 2^4 + \boldsymbol{1} \cdot 2^3 + \boldsymbol{1} = 89_{10}$.

A further problem arises from the basis of representation; generally, the insertion of numeric values through an interface or shell is done using the decimal base, which is naturally used by humans, while the internal representation of the computer is binary. This leads to the introduction of sneaky approximations: the decimal number 0.1 is already stored in an approximate way because its binary representation, $0.0\overline{0011}$, is periodic.

[1] `https://www.sagemath.org/`

Example 2. *Now, how can we represent* 0.1_{10} *in binary? We can first observe that* $0.1 = \frac{1}{10}$*: given that* $10_{10} = 1010_2$*, we can now represent this fraction as the following binary fraction:* $\frac{1}{\mathbf{1010}}$*. Now, we can proceed with a binary division:*

$$
\begin{aligned}
\frac{\mathbf{1}}{\mathbf{1010}} \cdot \mathbf{1} &= \boldsymbol{0} \cdot 1 \\
\frac{10}{1010} \cdot \frac{1}{10} &= \boldsymbol{0} \cdot \frac{1}{10} \\
\frac{100}{1010} \cdot \frac{1}{100} &= \boldsymbol{0} \cdot \frac{1}{100} \\
\frac{1000}{1010} \cdot \frac{1}{1000} &= \boldsymbol{0} \cdot \frac{1}{1000} \\
\frac{10000}{1010} \cdot \frac{1}{10000} &= \boldsymbol{1} \cdot \frac{1}{10000} \\
\frac{1100}{1010} \cdot \frac{1}{100000} &= \boldsymbol{1} \cdot \frac{1}{100000} \\
\frac{100}{1010} \cdot \frac{1}{1000000} &= \boldsymbol{0} \cdot \frac{1}{1000000}
\end{aligned}
\qquad
\begin{aligned}
&+ \frac{10}{1010} \cdot \frac{1}{10} \\
&+ \frac{100}{1010} \cdot \frac{1}{100} \\
&+ \frac{1000}{1010} \cdot \frac{1}{1000} \\
&+ \frac{10000}{1010} \cdot \frac{1}{10000} \\
&+ \frac{1100}{1010} \cdot \frac{1}{100000} \\
&+ \frac{100}{1010} \cdot \frac{1}{1000000} \\
&+ \frac{1000}{1010} \cdot \frac{1}{10000000}
\end{aligned}
$$

If we now continue to perform such operation and we read the quotient of the division in bold, we will se that we will obtain a number with infinite digits: $0.0001100110011\ldots$*. Given that the* 0011 *digits are repeating, we can now represent such number with the periodic notation, that is* $0.0\overline{0011}$*. This implies that the number that* 0.1 *will be represented in main machine as* $\approx 0.0999999\ldots$*.*

The basis representation is at the foundation of the finite number representation, and therefore also to the IEEE floating-point representation. We shall now provide yet another formal definition for defining the set of finite numbers that can be represented with precision of t β-digits.

Definition 2 (Finite Numbers)**.** *The finite number system* $F_{(\beta,t,\lambda,\omega)}$ *[7, 10] represents the set of all the possible numbers representable in a basis* $\beta \geq 2$ *as in Definition 1 with* t *significant digits and a interval bound over the exponent* $p \in [\lambda, \omega]$*:* $F_{(\beta,t,\lambda,\omega)} := \{0\} \cup \left\{ \alpha \in \mathbb{R} \;\middle|\; \alpha = \pm \left(\sum_{i=0}^{t-1} \alpha_i \beta^{-i} \right) \beta^p, \lambda \leq p \leq \omega \right\}$ *When, after a computation, the resulting number has* $p < \lambda$*, then we will have a* ***underflow*** *and, if* $p > \omega$*, we will have a* ***overflow****: these numbers will be respectively to small and too large to be represented in the number set.*

Let us now discuss how such definition is linked with the actual machine representation. Given that modern computers represents numbers via bits, the need to represent numbers with $\beta = 2$. Then, single float arithmetic represents numbers using $n = 32$ bits: such bits are distributed among the sign bit s, t mantissa digits and d exponent digits that can have $\omega - \lambda + 1$ possible values. As showed in Figure 2, we need to use $d = 8$ bits for the *exponent* and $t = 23$ bits for the *mantissa*.

With respect to the *exponent*, this means that we can represent $\beta^d = 2^8 = 256$ possible exponents. In particular, we reserve numbers $p = 0$ and $p = 255$ for representing the following numbers:

- If $p = 255$ and $m \neq 0$, then the number represents a numerical error `NaN` (e.g., division by zero).
- If $p = 255$ and $m = 0$, then the number represents $\pm\infty$, where $\pm$ is determined by the sign bit.
- If $p = 0$ and $m = 0$, then the number represents 0.
- If $p = 0$ and $m \neq 0$ then we have a not-normalized (or subnormal) number (see *mantissa*).

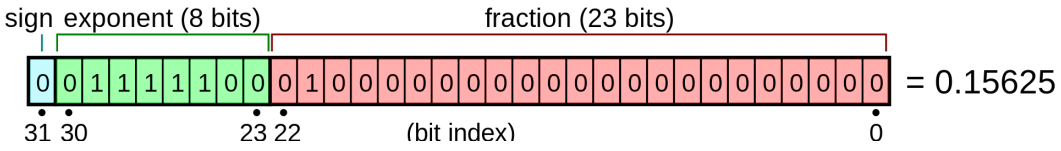


Figure 2: IEEE single floating point representation of the *normalized* float 0.15625. [en.Wikipedia]

All the remaining numbers are called *normalized* floats: we are now left with 254 possible exponents in the range $R = [1, 254] \subseteq \mathbb{Z}$: after evenly splitting the exponents at the middle, we will be able to represent exponents in the interval $N = [-126, 127] \subseteq \mathbb{Z}$. Then, we shall define a bias value $b = 127$ such that we are able to map the machine representation of the number of the exponent in R to the actual value in N: we can then define a function $f(r) = r - b$, where $f : R \to N$.

With respect to the 23 bits of the *mantissa*, they represent the decimal part of a binary number (see Example 2), where the unit might be either 1 or 0. For *normalized* floats, the unit is assumed to be 1, while for the non-normalized one it is 0 [2]. We can see that this requirement matches with the formula provided in Definition 2: we will assume that $t = 23$ with $\alpha_0 \in \{0, 1\}$ and, given that the last digit of the mantissa (22) represents the most significant digit of the decimal representation (2^{-1}), we will have that all the mantissa digits from 22 to 0 will be stored in $\{\alpha_1, \ldots, \alpha_{23}\}$.

Example 3. *With reference to Figure 2, we now want to convert the IEEE representation to the associated real number. Our exponent is* $r = 1111100_2 = 124_{10}$*: given that* $124 \notin \{0, 255\}$*, our float is normalized and our associated exponent is* $b = f(r) = 124 - 127 = -3$*. Our mantissa is* $m = \mathbf{1}.01000000000000000000000_2$ *with* $\alpha_1 = \mathbf{1}$*: in decimal representation, this means that* $m = 1_2 + 0.01_2 = 1_{10} + 2^{-2} = 1.25$*. Finally, we'll have that the resulting number will be:*

$$\underbrace{0}_{s}\ \underbrace{01111100}_{r}\ \underbrace{01000000000000000000000}_{m'=m-1.0}\ \underbrace{\qquad}_{\texttt{IEEE}} = -1^s \cdot (m) \cdot \beta^{f(r)} = 0.15625$$

Please observe that, for any possible not-normalized number, the exponent will be always $b = -126$ *by definition and, given that* $\alpha_1 = 0$, $m' = 0$ *[7]. Therefore, we might observe that normalized and not-normalized numbers will describe two different set of finite numbers.*

At this stage, a bit of coding will clarify all the former mathematical definitions. In order to represent a decimal number either as a float (Line 3) or decomposed into sign, exponent and mantissa (Line 6 downwards), we can use the code provided by the `GNU C Library` in the `ieee754.h` file. For cross-platform compatibility issues, the code is also provided in the associated source code. This double representation is possible via the `ieee754_float` union[2], where both a `float` f and a bit field[3] share the same memory location. With reference to the little endian architecture, we will know that the first 23 bits will belong to the mantissa, followed by the 8 bits of the exponent, and last the bit of the sign. For convenience of representation, we shall associate such bit sequences to unsigned integers.

```
union ieee754_float
{
    float f;

    /* This is the IEEE 754 single-precision format. */
    struct  {
#if   __BYTE_ORDER == __LITTLE_ENDIAN
unsigned int mantissa:23; // The remaining 23 bits are used to represent the mantissa
unsigned int exponent:8; // The subsequent 8 bits are assigned to the exponent
unsigned int sign:1;// The first bit, is assigned to the sign
#endif          /* Little endian. */
#if   __BYTE_ORDER == __BIG_ENDIAN
// Big endians represent the numbers similarly, but in the reverse direction.

unsigned int sign:1;
unsigned int exponent:8;
unsigned int mantissa:23;
#endif          /* Big endian. */

// Implementation specific features: computation errors results (e.g., divide by zero).
bool NaN() const { return exponent == 255 && mantissa != 0; }
// Maximum exponent to zero mantissa is used to represent the infinity
bool isInfinity() const { return exponent == 255 && mantissa == 0; }
// Zero mantissa and zero exponent represent the actual zero value.
bool isZero() const { return mantissa == 0 && exponent == 0; }
    } parts;
};
```

[2] A union is a user defined type where all the fields share the same memory location: as a consequence, the `sizeof` of the union will be equal to the field having the largest `sizeof`.

[3] A bit field is a data structure consisting of adjacent fields which have been allocated to hold a specific sequence of bits.

We can also define an utility macro, that is going to ease the definition of a new `ieee754_float` and initialize it with a float value:

```
#define declare_printable_float(name, val) ieee754_float name = { .f = (val) }
```

At this point, we can create a new variable `fl` containing the value 0.15625 as follows:

```
declare_printable_float(fl, 0.15625f);
```

Example 4. *At this point, we want to override the* ***`operator<<`*** *and print all the components of the mantissa. Before doing so, we must observe that the mantissa representation of* ***`ieee754_float`*** *is not a decimal, and therefore* ***`fl.parts.mantissa`*** $\neq m$ *as defined in Example 3. With reference to Example 1, the* ***`unsigned int`*** *associated to* $A =$ `01000000000000000000000` *is* $01000000000000000000000_2 = 2097152_{10} = 2^{21}$*. We can observe that, if we divide such number by* 2^{23} *and then sum it by* 1.0*, then we will obtain the expected number of* m *for normalized float, that is* $m = 1 + \frac{A_{10}}{2^{23}}$*. The final formula required to provide the decimal representation from a* ***`ieee754_float`*** *named* ***`fl`*** *will be the following:*

$$\texttt{printNormalized}(fl) := (-1)^{\texttt{fl.parts.sign}} \cdot (1.0 + {}^{\texttt{fl.parts.mantissa}}/_{2^{23}}) \cdot 2^{\texttt{fl.parts.exponent}-127}$$

With reference to Example 3, the formula for the not normalized numbers will then become:

$$\texttt{printNotNormalized}(fl) := (-1)^{\texttt{fl.parts.sign}} \cdot ({}^{\texttt{fl.parts.mantissa}}/_{2^{23}}) \cdot 2^{-126}$$

Now, we are ready to override the `operator<<` to print the information of the binary representation, and then convert it back to a float number by using some mathematical computation. In order to pretty print the information, we're going to use `libfort`[4], which has been already included in the project as a git submodule (Line 1). Such definition immediately follows an utility function that is going to print all the numbers with at most 26 decimals (Line 8): as you might see from Line 33, we adopt the same equation from Example 4 for reconstructing the float information from the bit fields decomposing the float representation.

Please note that the "␣" character is used to explicit show how many spaces are introduced in the strings.

```
1  #include <fort.hpp>
2  #include <bitset>
3
4  #include <utils/float.h>
5  #include <cmath>
6  #include <iomanip>
7
8  template <typename T> std::string tostr(const T& t) {
9      std::ostringstream os;
10     os << std::setprecision(26) << t;
11     return os.str();
12 }
13
14 std::ostream &operator<<(std::ostream &os, ieee754_float &classe) {
15     /*
16      * Checking if the number is normalized as defined by the IEEE754 standard
17      */
18     bool isNumberNotNormalized = ((classe.parts.exponent == 0) && (classe.parts.mantissa != 0.0));
19
20     /*
21      * Using the utility methods from the parts struct to print the final string result and the
22      * exponent value
23      */
24     std::string result;
25     std::string exponent;
26     if (classe.parts.NaN()) {
27 result = "NaN";
28     } else if (classe.parts.isZero()) {
29 result = "0";
30     } else if (classe.parts.isInfinity()) {
```

[4] https://github.com/seleznevae/libfort

```
result = "∞";
    } else {
result = tostr(isNumberNotNormalized ?
  (classe.parts.mantissa / std::pow(2, 23)) * std::pow(2, -126) :
  (1.0f + classe.parts.mantissa / std::pow(2, 23)) * std::pow(2, classe.parts.exponent - 127)
      );
exponent = (isNumberNotNormalized ? "2^-126" : "2^"+std::to_string(classe.parts.exponent-127));
    }

    /*
     * Bitsets allow to ingest a given variable and to fill a set of bits. Then, the set of bits can
     * be printed to a string using an utility function. Before doing that, we need to typecast the
     * float into a unsigned number. The most effective way to do this is to typecast tye float
     * pointer &classe.f to a unsigned long ponter, and then return the associated value by
     * dereference. Then, we need to tell the class that the final representation will be
     * sizeof(float)*CHAR_BIT bits long.
     */
    std::bitset<sizeof(float)*CHAR_BIT> x(*reinterpret_cast<unsigned long*>(&classe.f));

    // Using LibFort for pretty printing the
    fort::char_table table;
    // Setting the table's borders
    table.set_border_style(FT_SOLID_ROUND_STYLE);
    // Setting the style for the first column
    table.column(0).set_cell_text_style(fort::text_style::bold);

    // First Column // Second Column // Line End
    table << "Number" << tostr(classe.f) << fort::endr
  << "IsNormalized" << ((!isNumberNotNormalized) ? "true" : "false") << fort::endr
  << "Sign"  << (classe.parts.sign) << fort::endr
  << "Exponent" << classe.parts.exponent << fort::endr
  << "Mantissa" << classe.parts.mantissa << fort::endr
  << "Binary␣Representation" << x << fort::endr
  // Bitsets correctly handle unsigned ints, and so we need only to specify how many bits are
  // represented
  << "Binary␣Sign" << std::bitset<1>(classe.parts.sign) << fort::endr
  // Adding some trailing space to align the binary representation of the exponent to the
  // binary representation of the float
  << "Binary␣Exponent" << "␣"+std::bitset<8>(classe.parts.exponent).to_string() << fort::endr
  << "Binary␣Mantissa" << "␣"+std::bitset<23>(classe.parts.mantissa).to_string() << fort::endr
  << "Numeric␣Exponent" << exponent << fort::endr
  << "Result" << (classe.parts.sign ? "-" : "+") + result << fort::endr;

    return os << std::fixed << table.to_string() << std::endl;
}
```

Now, we can print a bit of float numbers and see how they are represented in IEEE754 format (see function `printing_float_representation()` from the code base).

1.2 Machine Epsilon and Floating Point Arithmetic

Given a specific finite number set $F_{(\beta,t,\lambda,\omega)}$, we are now interested to get which is the smallest error that such representation can commit in approximating real numbers $\mathbb{R}$. In particular, we might define two kind of approximations [6]:

- **truncation** of α to the t-th digit: $fl_T(\alpha) = \pm \left(\sum_{i=0}^{t-1} \alpha_i \beta^{-1} \right) \beta^p$
- **rounding** of α to the t-th digit: $fl(\alpha) = \pm fl_T \left(\left(\sum_{i=0}^{t} \alpha_i \beta^{-i} + \beta/2 \beta^{-t} \right) \beta^p \right)$

We might prove that the following results hold:

Lemma 1. *For each $\alpha \in \mathbb{R}$, both $|fl_T(\alpha) - \alpha| < \beta^{p-t}$ and $|fl(\alpha) - \alpha| \leq \frac{1}{2}\beta^{p-t}$ hold for $F_{(\beta,t,\lambda,\omega)}$.*

As a consequence, we might represent the machine epsilon as the smallest possible error that the machine can make; given that the *floating-point units* (FPU) of each CPU might have different precision values, we might consider the previous lemma as providing a lower bound for the smallest possible error:

Definition 3 (Machine Epsilon)**.** *The* ***machine epsilon*** *for a specific finite number set $F_{(\beta,t,\lambda,\omega)}$ is a quantity defined as half the distance between 1 and the next larger floating point number, that is $\epsilon \geq \frac{1}{2}\beta^{1-t}$ and that satisfies the following proposition:*

$$\forall \alpha \in \mathbb{R}\backslash\{0\}.\exists \tilde{\alpha} \in F_{(\beta,t,\lambda,\omega)}. \left|\frac{\alpha - \tilde{\alpha}}{\alpha}\right| \leq \epsilon$$

Id est, ϵ is the ***relative error*** *for representing the real number $\alpha \neq 0$ as $\tilde{\alpha}$.*

If we run the following code over an Intel® Xeon® CPU E3-1505M v6 3.00GHz×8 processor, then we will definitively see that its machine epsilon matches with the former definition:

```
// nextafterf returns the next represetable float following 1.0f.
// 2 is given as an upper bound.
float from1 = 1.0, to1 = nextafterf(from1, 2);
std::cout << "The␣next␣representable␣float␣after␣" << from1 << "␣is␣" << to1 << std::endl;
std::cout << std::numeric_limits<float>::epsilon() << std::endl;
assert(std::numeric_limits<float>::epsilon() == (to1 - 1.0f));
declare_printable_float(toPrint, (to1 - 1.0f));
std::cout << toPrint << std::endl;
```

We might also prove that the former proposition defining ϵ can be rewritten in terms of rounding fl as follows:

Lemma 2. *The difference between a real number α and its closest floating point approximation $fl(\alpha)$ is always smaller than ϵ in relative terms. Id est:*

$$\forall \alpha \in \mathbb{R}.\exists u \leq \epsilon.\ fl(\alpha) = \alpha(1+u)$$

1.3 Error Propagation in Floating Point Arithmetic

Beyond the errors introduced by the individual operations, more problems arise while performing a sequence of numeric operations where, at each step, previously calculated values are used as inputs. In this case, an error propagation occurs, the magnitude of which is very often far from negligible. The control and management of this phenomenon are essential in all computer applications implementing theoretical mathematical models with some approximations. Error propagation theory provides some essential tools for this purpose.

In Section 1, we introduced the concept of a *mathematical model* and of its *computational representation*. More formally, we can define a mathematical problem as a function $f\colon X \to Y$ from a space of the data to a space of the solutions; similarly, a *computational representation* of such mathematical model is another function $\tilde{f}\colon X \to Y$ between the same two spaces [10]. Except in trivial cases, $\tilde{f}$ cannot be continuous or will contain mathematical operations that are not numerically robust. To evaluate the extent of error propagation in a sequence of numerical operations, we need to consider the **total error** given by the sum of the individual errors. Given an exact function f, we introduce a relative error factor representing the approximation introduced by finite arithmetic over a specific finite set of numbers via $\tilde{f}$. The following quantity indicates the extent of such relative error: $\left|\frac{f(x)-\tilde{f}(x)}{f(x)}\right|$. We might then say that $\tilde{f}$ is **accurate** if its result is in $O(\epsilon)$.

Last, we shall define when a floating point operation $\circledast$ is a good implementation of a mathematical (binary) operation $*$:

Axiom 1 (Fundamental Axiom of Floating Point Arithmetic)**.** *Every operation of floating point arithmetic $\circledast$ is exact up to a relative error of size at most ϵ [10]:*

$$\forall \tilde{\alpha}, \tilde{\alpha}' \in F_{(\beta,t,\lambda,\omega)}.\exists u \leq \epsilon.\ \tilde{\alpha} \circledast \tilde{\alpha}' = (\tilde{\alpha} * \tilde{\alpha}')(1+\epsilon)$$

Under the assumption that a machine has a good implementation of the floating-point arithmetic via its FPU, we might now prove that, while multiplication and division are accurate, sum and subtraction are not accurate for $\tilde{\alpha} \pm \tilde{\alpha}' \to 0$

Lemma 3. *Multiplication and division are accurate.*

Algorithm 1 Iterative (and wrong) implementation of x^y for $y \in \mathbb{N}$

```
1: function IterativeIntegerPow(x,y)
2:     if y≤ 0 then return 1.0
3:     else if y= 1 then return x
4:     else
5:         result ← 1.0
6:         for i ← 1 to y do
7:             result ← result · y
8:         end for
9:         return result
10:    end if
11: end function
```

Proof. Given $x, y \in \mathbb{R}$, we can see that (and similarly for division):

$$\frac{fl(fl(x) \cdot fl(y)) - xy}{xy} = \frac{x(1+u_1)y(1+u_2)(1+u_3) - xy}{xy} \approx \frac{x(1+u)y(1+u)(1+u) - xy}{xy}$$
$$= \frac{xy + xy(3u + 3u^2 + u^3) - xy}{xy} \in O(u)$$

□

Lemma 4. *Sum and substraction are not accurate when* $|x \pm y| \to 0$

Proof. Given $x, y \in \mathbb{R}$, we can see that (and similarly for division):

$$\frac{fl(fl(x) \pm fl(y)) - (x \pm y)}{x \pm y} = \frac{(x(1+u_1) \pm y(1+u_2))(1+u_3) - (x \pm y)}{x \pm y}$$
$$\approx \frac{x}{x \pm y}u + \frac{y}{x \pm y}u + u$$

We observe that for $x \pm y \to 0$, then $\frac{x}{x \pm y}u + \frac{y}{x \pm y}u \gg u$: in this case, the error is not due to the specific arithmetic operation, but it is due to the floating-point representation of x and y, which representation error is amplified by the sum and subtraction operations. We refer to this problem as **numerical cancellation**. □

As a consequence of the former lemma, a more numerical stable algorithm $\tilde{f}$ for f will be a function reducing the possible number of floating-point arithmetic operations, so that the errors are minimized.

Example 5. *Suppose that you want to implement an algorithm for* x^y *for* $y \in \mathbb{N}$*: given that* x^y *is mathematically defined as* $\prod_{1 \le i \le y} x$*, it is tempting to define a* `pow` *operator using an iterative approach (Algorithm 1). On the other hand, if we assume to have numerical stable implementations for both the* e^x *and* $\ln(x)$*, then we can use some simple math rewriting to obtain:*

$$x^y = e^{\ln(x^y)} = e^{y \ln(x)}$$

By doing so, we reduce the number of floating-point arithmetic operations to a constant value that does not depend by y*. You can see an implementation for such a function following a similar approach at:* `http://www.netlib.org/fdlibm/e_pow.c`*.*

As an exercise, we might as well try to change the definition of the *Binet's Formula* from the former tutorial such that the power of ϕ is evaluated in an iterative way. After moving the previously defined lambda function to a function named `binet_formula`, we can define a new function `binet_formula_error` where the usage of the numerically stable `std::pow` function is replaced by an iterative version. The final source code should look something like this:

```
float iterative_pow(float base, int pow) {
    float intermediate = 1.0;
    if (pow <= 0)
```

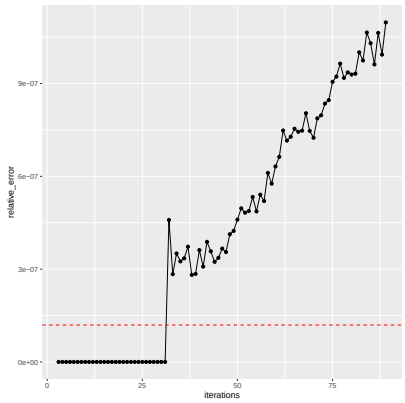


Figure 3: Total error plot for `binet_formula_error`. The red dashed line represents the machine epsilon for Intel® Xeon® CPU E3-1505M v6 3.00GHz×8 processor, i.e. $1.19209290 \cdot 10^{-7}$

```
return 1.0;
    if (pow == 1)
return base;
    for (int i = 1; i<=pow; i++)
intermediate *= base;
    return intermediate;
}

unsigned long binet_formula_error(unsigned long n) {
    float fib = (iterative_pow(((1.0f + (float)sqrtf(5.0f))*0.5f), n) - iterative_pow(( 1.0f - ((1.0f
        ↪ + (float)sqrtf(5.0f))*0.5f)), n))/sqrtf(5);
    return roundf(fib);
}

void total_error_plot() {
    for (int i = 3; i<90; i++) {
std::cout << "i=" << i << "␣-->␣" << ((double)std::abs((long long)binet_formula_error(i) - (long long)
    ↪ binet_formula(i)))/((double)binet_formula(i)) << ",␣";
    }
}
```

If we now plot the relative error on the y axis and the Fibonacci year (also corresponding to the number of the power iterations) in the x axis, then we might obtain a plot similar to the one in Figure 3: for $i \geq 32$, we observe that the numerical error is greater than the machine epsilon, thus making the specific implementation of the `binet_formula_error` not stable for values greater than 32.

In the following sections, we're going to analyse several different techniques through which we can cope with such rounding errors.

2 Interval Arithmetic

As you might recall from page 4 and from your previous knowledge of *mathematical analysis*, a **closed interval** $[a, b]$ represents a range of numbers from a to b, including the end values. More formally:

Definition 4 (Closed Interval)**.** *A closed interval $[a, b]$ over a partially odered set $(S, \preceq)$ is defined as the following set of elements:*

$$[a, b] := \{\, x \in S \mid a \preceq x \preceq b \,\}$$

We can also define the following shorthand:

$$x \in [a, b] \Leftrightarrow a \preceq x \preceq b$$

In our case, we're going to consider closed intervals over the real numbers, and therefore we're going to consider the partially ordered set $(\mathbb{R}, \leq)$ or $(F_{(\beta,t,\lambda,\omega)}, \leq)$.

The need to enclose a number arises from measure theory: if we want to measure a linear length with a ruler that can measure dimensions greater than a millimeter, the ruler's **observational error**

will be ± 0.1cm. Therefore, any measured length of x can be represented as a closing interval $[x - 0.1\text{cm},\ x + 0.1\text{cm}]$. With respect to the Newton's law $F = ma$, given that both m and a can be measured with tools having specific observational errors, F will be also affected by an observational error, and therefore can be represented as a closed interval.

Nevertheless, the act of using intervals for bounding the minimum and maximum possible representation value might seem week: if an operation (or generally, an algorithmic implementation) is not accurate, we will still fail to get the actual solution itself. As seen in `binet_formula_error`, the roundoff error introduced in Section 1.2 on page 6 can destroy a numerical solution if it manages to accumulate sufficiently. Moreover, we can show that repearing a calculation with increased (not infinite) precision does not necessarily provide increasingly accurate results [5], due to the involved mathematical operations. Therefore, when it is "prohibitively difficult" to tell where we need to ensure accuracy, we might forechoose to define narrow closed interval instead of computing a numerical approximation using finite numbers and then worrying about the final accuracy.

We can now define some common arithmetic operators as follows:

$$[a, b] + [c, d] := [a + c, b + d]$$

$$[a, b] - [c, d] := [a - d, b - c]$$

$$[a, b] \cdot [c, d] := [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$$

$$[a, b]/[c, d] := [a, b] \cdot \left[\frac{1}{d}, \frac{1}{c}\right] \qquad 0 \notin [c, d]$$

$$[a, b]^2 := \begin{cases} [0, \max(a^2, b^2)] & 0 \in [a, b] \\ [\min(a^2, b^2), \max(a^2, b^2)] & \text{oth.} \end{cases}$$

[9, 5] show the general idea for implementing other arithmetic operations over interval arithmetic.

The main weakness of interval arithmetic is that the resulting intervals might become larger than we would expect. This problem is due to the *dependency problem* [2]: if a variable occurs more than once in an expression, the computed result might not tightly bound the true value range of the expression. In fact, each time a variable occurs in an expression, it potentially makes the interval larger. A simple solution will be to rewrite the mathematical function of choice, so that the variable only occurs once.

We can now show a simple scenario of such situation using once again some C++ coding. First, we can define a `struct IntervalArithmetic` containing the lower (`x`) and upper (`y`) bound of the interval.

```
#include <algorithm>
#include <cmath>
#include <stdexcept>
#include <ostream>

struct IntervalArithmetic {
    float x, y;
    /**
     * Defining the interval [x, y]
     * @param x minimum value
     * @param y maximum value
     */
    IntervalArithmetic(float x, float y);

    /**
     * Defining the interval over one single point
     * @param c
     */
    IntervalArithmetic(float c);
    IntervalArithmetic &operator+=(const IntervalArithmetic &i);
    IntervalArithmetic &operator-=(const IntervalArithmetic &i);
    IntervalArithmetic &operator*=(const IntervalArithmetic &i);
    IntervalArithmetic &operator/=(const IntervalArithmetic &i);
    IntervalArithmetic operator+(const IntervalArithmetic &i) const;
    IntervalArithmetic operator-(const IntervalArithmetic &i) const;
```

```
    IntervalArithmetic operator*(const IntervalArithmetic &i) const;
    IntervalArithmetic operator/(const IntervalArithmetic &i) const;
    IntervalArithmetic pow(unsigned int pow);
    friend std::ostream &operator<<(std::ostream &os, const IntervalArithmetic &arithmetic);
};
```

Now, we can override the * and define a pow function for such intervals.

```
IntervalArithmetic IntervalArithmetic::operator*(const IntervalArithmetic &i) const {
    IntervalArithmetic result = *this;
    result *= i;
    return result;
}

IntervalArithmetic &IntervalArithmetic::operator*=(const IntervalArithmetic &i) {
    x = std::min({x*i.x, x*i.y, y*i.x, y*i.y});
    y = std::max({x*i.x, x*i.y, y*i.x, y*i.y});
    return *this;
}

IntervalArithmetic IntervalArithmetic::pow(unsigned int pow) {
    if ((x <= 0) && (0 <= y)) {
return {0, (float)std::max(std::pow(x, pow), std::pow(y, pow))};
    } else {
return {(float)std::min(std::pow(x, pow), std::pow(y, pow)), (float)std::max(std::pow(x, pow), std::
    ↪ pow(y, pow))};
    }
}
```

The other operators to override are left as an exercise[5]. After overriding all the mathematical operators of interest in C++, we can then define most of our mathematical functions as template functions, so that both floats and closed intervals can be used in these computations. We can now define the $f_1(x) := {}^{x}/_{x-1}$ as follows:

```
template <typename T> T fprime(const T &x, const T& one) {
    return x / (x - one);
}
```

After observing that we can easily rewrite such function into another one where x appears only once, we can also define an equivalent function expressing the same mathematical concept:

$$f_2(x) := 1 + \frac{1}{x-1} = \frac{x-1+1}{x-1} = \frac{x}{x-1} = f_1(x)$$

```
template <typename T> T fsecond(const T &x, const T& one) {
    return one + one / (x - one);
}
```

We can now simply test that the first function might produce a less narrow interval than the second:

```
void examples_on_interval_arithmetic() {
    std::cout << fprime<IntervalArithmetic>({2.9, 3.1}, {1.0,1.0}) << std::endl; // [1.38095, 1.63158]
    std::cout << fprime(3.0, 1.0) << std::endl; // 1.5

    std::cout << fsecond<IntervalArithmetic>({2.9, 3.1}, {1.0,1.0}) << std::endl;// [1.47619, 1.52632]
    std::cout << fsecond(3.0, 1.0) << std::endl; // 1.5
}
```

2.1 Sphere-AABB Overlap Tests

Interval arithmetic can be also used in collision detection scenarios, where a given surface S can be represented as a mathematical equation. As per previous knowledge, we know that a bounding box that can be described by just one equation is the sphere. Therefore, given the mathematical equation

[5] See https://github.com/jackbergus/NotesOnProgramming2020/blob/master/src/numeric/structures/IntervalArithmetic.cpp for a solution.

of a sphere $S(\vec{p}) = (p_0 - c_0)^2 + (p_1 - c_1)^2 + (p_2 - c_2)^2 - r^2$ of radius of length $r \in \mathbb{R}$ and center in $\vec{c} = (c_0, c_1, c_2)$, we can straightforwardly represent it in interval arithmetic as follows:

$$\overline{S}(x, y, z) = (x - [c_0, c_0])^2 + (y - [c_1, c_1])^2 + (z - [c_2, c_2])^2 - [r, r]^2$$

Such sphere can be implemneted using the OpenGL Mathematics Library (GLM)[6] with the following definition:

```
class Sphere {
    glm::tvec3<float> center;
    float radius;
public:
    Sphere(const glm::tvec3<float> &center, float radius) : center(center), radius(radius) {}
    IntervalArithmetic equation(const glm::tvec3<float>& point);
    IntervalArithmetic equation(const IntervalArithmetic& cx,
   const IntervalArithmetic& cy,
   const IntervalArithmetic& cz
  );
};
```

By using the Interval Arithmetic, we can implement $\overline{S}$ on `Sphere` as follows:

```
IntervalArithmetic Sphere::equation(const glm::tvec3<float> &point) {
    IntervalArithmetic x{point.x};
    IntervalArithmetic y{point.y};
    IntervalArithmetic z{point.z};

    return equation(x, y, z);
}

IntervalArithmetic Sphere::equation(const IntervalArithmetic &x,
const IntervalArithmetic &y,
const IntervalArithmetic &z
      ) {
    IntervalArithmetic cx{center.x};
    IntervalArithmetic cy{center.y};
    IntervalArithmetic cz{center.z};
    IntervalArithmetic r{radius};

    return (x - cx).pow(2) + (y - cy).pow(2) + (z -cz).pow(2) - r.pow(2);
}
```

Given that for a given point $\vec{x} \in \mathbb{R}^3$ $S(\vec{x}) > 0$ iff. the point is outside the surface of the sphere, we can mimic the same behaviour in $\overline{S}$ and similarly check if $\min \overline{S}(\vec{x}) > 0$.

After reminding that we can represent an AABB in the following ways:

1. the minimum ($\vec{m}$) and maximum ($\vec{M}$) coordinate value
2. the minimum coordinate value and the diameter extents from this corner
3. the center coordinate and the half-width extents

we can easily observe that we can first represent an AABB as in option 1, and then transform such points in three intervals, $x_I = [m_0, M_0]$, $y_I = [m_1, M_1]$ and $z_I = [m_2, M_2]$ such that $(\vec{m}, \vec{M}) \equiv \langle x_I, y_I, z_I \rangle$. Now, the sphere-AABB overlap test reduces to checking whether $\min \overline{S}(x_I, y_I, z_I) \leq 0$. Finally, we can provide the implementation of the ABBB, where the overlap testing is also provided:

```
#include <vec3.hpp>
#include "Sphere.h"
// Intersection cases [2]
enum tagIntersectionTest {
    liesOutside,
    intersectsBoundary,
    liesFullyInside
};
```

[6] `https://glm.g-truc.net`. Please note that such library has been already imported as a submodule in the repository associated to the tutorial.

```
// Class Definition
class AABB {
    glm::tvec3<float> min, max;

public:
    AABB(const glm::tvec3<float> &m, const glm::tvec3<float> &M);
    tagIntersectionTest testWithSphere(Sphere& s);
};

//Implementation: Constructor
AABB::AABB(const glm::tvec3<float> &m, const glm::tvec3<float> &M) : min(m), max(M) {}

// Implementation: Test
tagIntersectionTest AABB::testWithSphere(Sphere &s) {
    const IntervalArithmetic &result = s.equation({min.x, max.x}, {min.y, max.y}, {min.z, max.z});
    if (result.x > 0)
return liesOutside;
    else if ((result.x <= 0) && (result.y <= 0)) {
return intersectsBoundary;
    } else if (result.y > 0)
return liesFullyInside;
    throw std::runtime_error("ERROR: no case was matched");
}
```

3 Gottschalk's Algorithm for OBB Overlap Tests

As per previous knowledge, an Oriented Bounding Box can be simply defined as an AABB with an arbitrary orientation, that can be expressed as a rotation matrix R^A. Similarly to the AABB scenario, we can define the OOBB in several different ways:

1. a collection of 8 vertices $\{\vec{r}_1, \dots, \vec{r}_8\}$.
2. a collection of 6 planes $\{\pi_1, \dots, \pi_6\}$.
3. a center point T^A, an orientation matrix R^A, three half-edge lengths $\vec{h}^A = (a_0, a_1, a_2)$.

We might observe that the most space-sparing representation is the last one, which is also the one similar to the 3-rd possible definition of an AABB. As previously outlined in [3], we can check two main classes of algorithms that can be adopted for implementing OBB overlap tests:

- **Linear Programming**: a *plane* π can be described by a position vector $\vec{r}_0 \in \pi$ and a non-zero orthogonal vector $\vec{n} \perp \pi$, π is defined as the set of points $\vec{r} \in \mathbb{R}^3$ that satisfy the following equation:

  $$\pi(\vec{r}) = \mathbf{0} \quad \text{s.t.} \quad \pi(\vec{r}) := \vec{n} \cdot (\vec{r} - \vec{r}_0)$$

  Therefore, an *half-space* can be described by $\pi(\vec{r}) \leq 0$. For two given OOBs described respectively by vertex sets $OBB_A = \{\vec{r}_1, \dots, \vec{r}_8\}$ and $OBB_B = \{\vec{r}'_1, \dots, \vec{r}'_8\}$, we want to find a **separating plane** π such that all the points in OBB_A line in a different half-space than the ones in OBB_B. This means that we need to solve the following system:

  $$\forall \vec{v} \in OBB_A . \pi(\vec{v}) < 0 \wedge \forall \vec{v}' \in OBB_B . \pi(\vec{v}') \geq 0$$

  If standard LP libraries can find a feasible solution, then the OBBs are disjoint. If no feasible solution can be found, then they overlap.

  Albeit this technique can rely on exiting efficient LP libraries, this problem implies solving a more general class of problems. Nevertheless, we can show that this problem can be trivially rewritten in a more simple formulation, thus making the whole resulting computation not so efficient.

- **Distance Computation**: overlap tests such as GJK [2, 4] and Lin-Canny [3] rely on the definition of a distance function δ over two OBBs for which $\delta(OBB_A, OBB_B) = 0$ iff. the two shapes overlap.

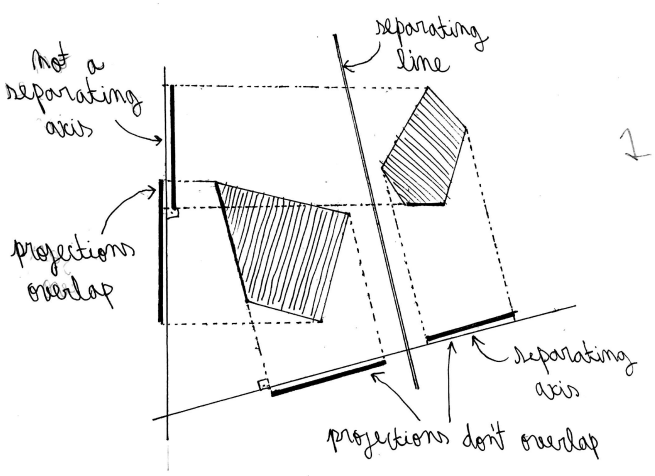


Figure 4: Graphical depiction of two 2D convex hulls separated by an axis (line). `https://hackmd.io/@US4ofdv7Sq2GRdxti381_A/ryFmIZrsl`

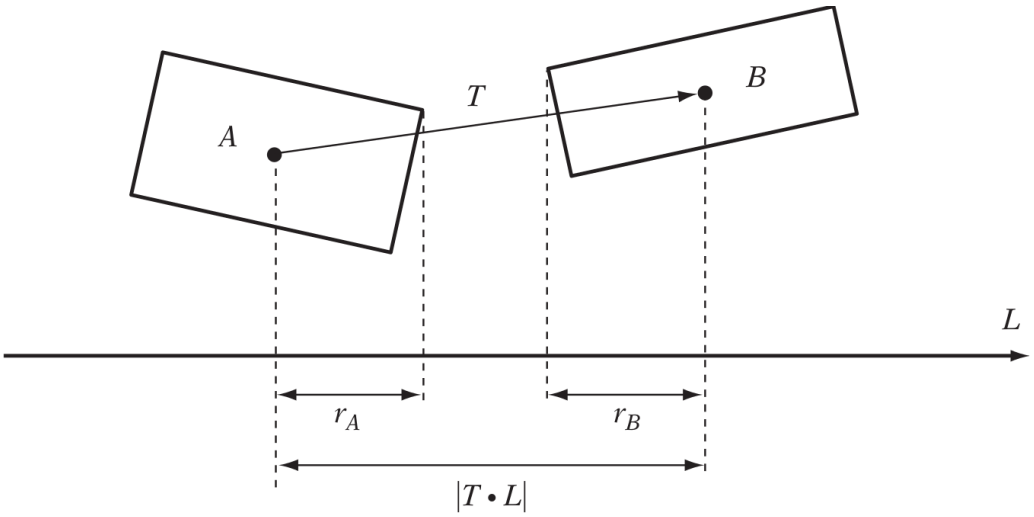


Figure 5: Gottschalk's formulation of the Separating Axis Theorem for OBBs over a specific axis L [2].

> Even in this case, such technique can be applied to even more general inputs than OBBs, such as convex polytopes: we want to show that we do not need to necessarily compute the distance between the two shapes, and therefore our computation can be made even more efficient.

The approach proposed in [3] uses the *hyperplane separation theorem* firstly introduced Hermann Minkowski (⋆1864 - †1909). This theorem is often simplified for the $\mathbb{R}^3$ vector space and referred as *separating axis theorem*: in fact, if a plane π is a separating plane, then any axis perpendicular to it is a separating axis. We need first a preliminary definition:

Definition 5 (Separating Axis)**.** *We say that an axis (or line) $\vec{n}$ is **separating** two point clouds A and B iff. the images of A and B under axial projection onto $\vec{n}$ are disjoint.*

Figure 4 provides an intuitive representation of such theorem for the 2D scenario. Now, we can enunciate the fol, where the lowing theorem:

Theorem 1 (Separating Axis Theorem)**.** *Two 3D convex hulls A and B are disjoint iff. there exists a separating axis L which is either perpendicular to a face of one of the 3D convex hulls, or is perpendicular to an edge of A or B.*

We can also show that, given two general convex polytopes with F faces and E edges, it is sufficient to test $E^2 + 2 \cdot F$ candidate axes to determine the overlap; for an OBB with $F = 6$ and $E = 12$, this would require 156 axes check. As per previous observations, we can reduce the number of comparisons to 15: we have only 3 distinct normals and 3 distinct face directions, thus reducing the number of the overall comparisons to the following candidate separating axes L: *(i)* the 3 column vectors (direction axes) of R^A, *(ii)*, the 3 column vectors (direction axes) of R^B, and *(iii)* the 9 cross product of such columns. We refer to [3] for additional details concerning the Separating Axes Theorem.

Let us assume that, for the moment, we are able to determine all the possible L candidate separate axes: for each one of these Ls (Figure 5), we can project an OBB_A into L as an interval $2r_i$, which width is the sum of the projection of any three mutually orthogonal edges. Therefore, the half-width r_i of such interval can be computed by summing up the half-edge lengths (a_0, a_1, a_2) of the three orthogonal edges represented as three vector columns R_0^A, R_1^A, R_2^A of the orientation matrix R^A. So, the length r_A becomes:

$$r_A = {}^1\!/_{|L|}(a_0|R_0^A \cdot L| + a_1|R_1^A \cdot L| + a_2|R_2^A \cdot L|)$$

Last, the box centers T^A and T^B project onto the the midpoint of the intervals obtained as per Definition 5 and Figure 4. The separation of the midpoints is equal to the length of the projection of the vector joining the box centers, that is $s = T \cdot \hat{L}$, where $\hat{L} = {}^L\!/_{|L|}$. Given that $T = T^B - T^A$, then the definition of s becomes:

$$s = {}^1\!/_{|L|}(|T^A - T^B| \cdot L)$$

Then, we can intuitively see from Figure 5 that the two OBBs are disjoint under the axial projection L iff the following condition holds:

$$s > r_A + r_B$$

Then, we can rewrite the final test as follows:

$$\begin{aligned} &|(T^A - T^B) \cdot L| \cdot {}^1\!/_{|L|} > \\ &\quad {}^1\!/_{|L|}(a_0|R_0^A \cdot L| + a_1|R_1^A \cdot L| + a_2|R_2^A \cdot L|) + \\ &\quad {}^1\!/_{|L|}(b_0|R_0^B \cdot L| + b_1|R_1^B \cdot L| + b_2|R_2^B \cdot L|) \end{aligned}$$

We can also observe that, when two OBB direction axes are parallel, they will provide a zero vector as a result of their cross product L. In addition to that, we want to avoid the division by zero and, for the previous statement, we can then multiply both sides of the inequality by $|L|$. This operation is possible because both *(i)* if such axes are parallel, there will be no separation axis, and therefore the test might as well be skipped, and *(ii)* zero norm axes will reduce to a zero length projection, leading to a $0 > 0$ test. Then, we can formulate the Separating Axis Theorem in the following formulation:

Lemma 5 (Gottschalk's Test)**.** *Two OBBs* $OOB_A = \left\langle T^A, R^A, \vec{h}^A \right\rangle$ *and* $OBB_B = \left\langle T^B, R^B, \vec{h}^B \right\rangle$ *overlap if there exist no separating axis* $L \in \{R_0^A, R_1^A, R_2^A, R_0^B, R_1^B, R_2^B, R_0^A \times R_0^B, \dots R_2^A \times R_2^B\}$ *satisfying the following equation:*

$$\begin{aligned} &|(T^A - T^B) \cdot L| > \\ &\quad (a_0|R_0^A \cdot L| + a_1|R_1^A \cdot L| + a_2|R_2^A \cdot L|) + \\ &\quad (b_0|R_0^B \cdot L| + b_1|R_1^B \cdot L| + b_2|R_2^B \cdot L|) \end{aligned} \tag{1}$$

In the following subsections, we are going to provide two different implementations of such test: one directly implementing the test as described the former lemma with some tricks, and the latter more optimized version reducing the number of floating-point arithmetic operations and removing all the possible operations that might lead to numerical cancellation. In all of these scenarios, we will find alternative approaches to interval arithmetic to solve these problems.

3.1 Naïf Test

As per previous lemma, the basic definition of the Gottschalk's test requires to perform the cross product between the OBBs' local axes. when these axes are nearly parallel, the result is a (nearly zero) vector, and all the projections are therefore zero. As a first approximation, we can assume

Algorithm 2 Robust Cross Product for the Gottschalk's Test [2].

1: **function** SepAxisCrossProd($\vec{ab}, \vec{cb}$) ▷ $\vec{ab} := \vec{b} - \vec{a}$ $\vec{cd} := \vec{d} - \vec{c}$
2: $\vec{mp} = \vec{ab} \times \vec{cb}$
3: **if not** `isZeroVector`($\vec{mp}$) **then**
4: **return** $\langle \vec{mp}, \texttt{true} \rangle$
5: **else**
6: $\vec{ca} = c - a$
7: $\vec{n} = \vec{ab} \times \vec{ca}$
8: $\vec{m} = \vec{ab} \times \vec{n}$
9: **if not** `isZeroVector`($\vec{m}$) **then return** $\langle \vec{m}, \texttt{true} \rangle$
10: **else return** $\langle \mathbf{0}, \texttt{false} \rangle$
11: **end if**
12: **end if**
13: **end function**

that any number that is lower than the machine epsilon resulting from a numerical linear algebra computation might be affected by a big error, and therefore we can define this macro to test whether the vector shall be considered as a near zero vector:

```
/**
 * Approximates the fact that the vector is a zero vector with the fact that each component is less
     ↪ than epsilon
 */
#define isZeroVector(v) (glm::all(glm::lessThan(glm::abs(v), glm::vec3(std::numeric_limits<float>::
    ↪ epsilon()))))
```

This consideration is self evident from the following definition of the cross product between $\vec{u}$ and $\vec{v}$

$$\vec{u} \times \vec{v} := \vec{n} \|\vec{u}\| \|\vec{v}\| \sin\theta$$

where $\theta = \widehat{\vec{u}\vec{v}}$ and $\vec{n}$ is the norm vector perpendicular to the plane where both vector lie. In fact, $\lim_{\theta \equiv \widehat{\vec{u}\vec{v}} \to 0} \sin\theta = 0$: numerically, this means that $\|\vec{u} - \vec{v}\| \approx \epsilon$ and, given the definition of the vector product in terms ov vector components:

$$\vec{u} \times \vec{v} := (u_1 v_2 - u_2 v_1,\ -(u_0 v_2 - u_2 v_0),\ u_0 v_1 - u_1 v_0)$$

E.g., if we assume that $\vec{u} = (a, b, c)$ and $\vec{v} = (a+\varepsilon, b-\varepsilon, c+\varepsilon)$ with $0 < \varepsilon \ll 1$, then the resulting cross product will become $(b\varepsilon + c\varepsilon,\ -(a\varepsilon + c\varepsilon),\ -(a\varepsilon + b\varepsilon))$. As per Lemma 4 on page 8, we can see that all the vector dimensions will suffer from numerical cancellation. Albeit we cannot provide a generalized cross product robust algorithm that will fit any use case where the cross product is required, we can find a specific cross product algorithm for the Gottschalk's Test.

Algorithm 2 provides the desired implementation: if the cross product between two vectors is not a zero vector, then the edges are not parallel, and the resulting vector can be indeed used as a separating axis (Line 4); otherwise, try to generate an alternative separating axis: given that $\vec{ab}$ and $\vec{cd}$ are nearly parallel, they might lie in some plane π: therefore, we might first create a vector $\vec{n} \perp \pi$, for then obtaining a new candidate axis vector $\vec{mp}$ which is perpendicular to $\vec{ab}$ and lying in π. Therefore, this algorithm is trying to compensate the cross product error by generating a novel axis, as we cannot possibly change the cross product to anything else.

Example 6. *This example provides a graphical depiction (Figure 6) of the typical use case scenario where the aforementioned algorithm might be useful. Let us suppose to have two vectors, have* $\vec{ab} = (0.04, 2.49, 0)$ *and* $\vec{cd} = (0.22, 1.75, -0.32)$*, which cross product produces the near zero vector* $\vec{mp} = (-0.8, 0.01, -0.47)$ *with respect to our graphical representation: given that* $\vec{cd}$ *and* $\vec{ab}$ *lie in a same plane (in light blue), then we can define the vector* $\vec{ac} = c - a$ *via which we can compute the final axis vector* $\vec{m} = (-1.18, 0.02, -6.81)$*.*

Representing Vectors for the robust Cross Product Given that the SepAxisCrossProd requires that each vector shall be decomposed into a source and a destination node, we also need to define a novel `Vector` struct, and represent all of the OBB's vectors via the `Vector` struct. Therefore, we shall first define the `Vector` class as follows:

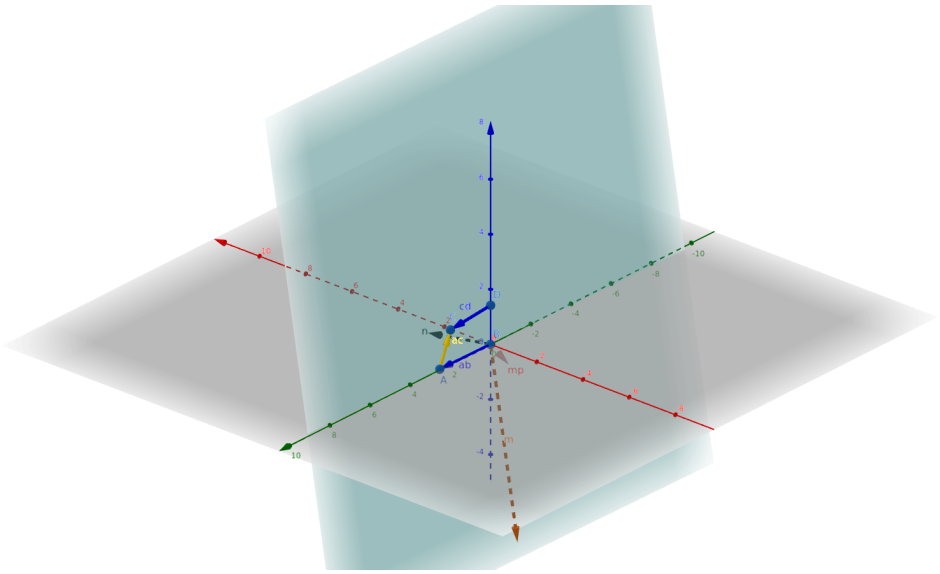


Figure 6: A use case scenario for the Robust Cross Product algorithm provided below, where mp is the first tentative cross product leading to a near zero vector, and m is the returned axis returning a non-zero vector.

```cpp
#include <glm.hpp>
#include <vec3.hpp>

struct Vector {
    bool hasPrecomputedResult;
    /**
     * The distinct representation from source and destination will be required from the robust cross
         ↪ product for the separate axis
     */
    glm::tvec3<float> source;
    glm::tvec3<float> destination;
    glm::tvec3<float> casted_difference;

    Vector(const Vector& rhs) = default;
    Vector& operator=(const Vector& rhs);

    /**
     * Defining a vector as the difference from point y to x.
     *
     * @param x Source Point
     * @param y Destination point
     */
    Vector(const glm::tvec3<float> &x, const glm::tvec3<float> &y);

    /**
     * Defining a vector as a direction.
     *
     * I.e., the destination point is exactly the direction vector, and the source point is the origin
         ↪  of the axes.
     *
     * @param direction Vector as a direction
     */
    Vector(const glm::tvec3<float> &direction);


    /**
     * Defining a vector as a component-wise direction.
     *
     * @param x
     * @param y
     * @param z
     */
    Vector(float x, float y, float z) : Vector{{x, y, z}} {};
```

```
    /**
     * Default constructor: zero vector.
     */
    Vector();



    /**
     * Returning the vector as the difference between two points. (Implicit casting)
     *
     * @return (destination point)-(source point)
     */
    explicit operator glm::tvec3<float>();

    /**
     * Explicit casting into a vector as the difference between two points
     * @return
     */
    glm::tvec3<float> asVector();

    /**
     * Returning the b-th component of the vector of its "actual" representation (as a point
         ↪ difference)
     * @param b    Vector position, counting from 0
     * @return     Value of the b-th component (if 0<=b<3).
     */
    float operator[](int b);
};
```

We can simply implement such definition as follows:

```
glm::tvec3<float> Vector::asVector() {
    if (!hasPrecomputedResult) { // Check whether we have a precomputed GLM representation
casted_difference = destination - source; // Represent the vector as a GLM vector
hasPrecomputedResult = true;
    }
    return casted_difference;
}

Vector::Vector(const glm::tvec3<float> &x, const glm::tvec3<float> &y) : source(x), destination(y),
     ↪ hasPrecomputedResult{false} {}

Vector::Vector(const glm::tvec3<float> &direction) : destination{direction} { source.x = 0.0f; source.
     ↪ y = 0.0f; source.z = 0.0f; hasPrecomputedResult = false; }

Vector::Vector() : Vector{0.0f, 0.0f, 0.0f} {}

Vector::operator glm::tvec3<float>() {
    return asVector();
}

float Vector::operator[](int b) {
    return asVector()[b];
}

Vector &Vector::operator=(const Vector &rhs) {
    hasPrecomputedResult = rhs.hasPrecomputedResult;
    source = rhs.source;
    destination = rhs.destination;
    casted_difference = rhs.casted_difference;
    return *this;
}
```

By using the last example as a reference, please observe that $\vec{m}$'s magnitude is far way bigger than $\vec{ab}$'s and $\vec{cd}$, and might as well further increase: this happens only because $\vec{ab}$, $\vec{cd}$, and $\vec{n}$ are not normalized vectors. A large magnitude ot the returned separating axis vector might also lead to a less accurate result because, as per definition of a Finite Number (Definition 2 on page 3), we can only represent each float with a limited number of bits. Given that the separating axis algorithm uses the local axes purely as direction vectors while the size of the edge is represented by the half-size, we might want to normalize such vectors before performing their cross product. Finally, we can provide

the required implementation for the algorithm as follows:

```
/**
 * This function normalizes the vector if and only if the provided vector is not a zero vector.
 *
 * @param x Vector to be normalized
 * @return Normalized vector
 */
glm::tvec3<float> normalizeNotZeroVector(const glm::tvec3<float> &x) {
    return (isZeroVector(x)) ? x : glm::normalize(x);
}


/**
 * Checks whether two vectors are parallel or not. If they are parallel, they have an intersection
 * axis (second argument true) that is provided by the first argument.
 * Otherwise, there is no separating axis.
 *
 * @param ab First vector
 * @param cd Second vector
 * @param doCrossNormalize Whether we need to normalize the cross product in the case that the vectors
     ↪  have a large magnitude
 * @return Returns a pair, where the second element returns whether the two vectors have a separating
     ↪ axis. If that element exists, then it is returned in the first argument of the pair.
     ↪ Otherwise, the first argument will be an arbitrary vector
 */
std::pair<glm::tvec3<float>, bool>
SepAxisCrossProd(Vector &ab, Vector &cd, bool doRobust) {
    // Determining the (numerical) vectors from the source and destination points;
    const glm::tvec3<float> vectorAB = ab.asVector(), vectorCD = cd.asVector();

    /*
     * When the two vectors are nearly parallel, the result of the cross product might be a zero
     * vector. As a consequence, all the projections over such axis will have a zero distance...
     */
    // NOTE:
    // =====
    //
    // Why are we normalizing the vectors for the cross product?
    // 1) Vectors having a large magnitude could also add significant errors while computing the cross
         ↪  product. If no bounding value for the magnitude of the vector is known, then it is a good
         ↪  idea to normalize the vector4
    // 2) Given that vectors in our scenario simply represent a direction, then we can ignore the
         ↪ original magnitude value.
    //
    // How to deal with the maximum expected magnitude will vary a lot depending on the scenario of
         ↪ choice and on the reason why we're computing the cross product.
    glm::tvec3<float> m = glm::cross(normalizeNotZeroVector(vectorAB), normalizeNotZeroVector(vectorCD
         ↪ ));
    if (!isZeroVector(m)) {
/*
 * Vectors (ab) and (cd) are not parallel: m is the potential separating axis. So, the second argument
     ↪  is true.
 */
return std::make_pair(m, true);
    } else if (doRobust) {
/*
 * ... unless we do an additional safety check: we need to test if the resulting cross product vector
     ↪ is also a (near) zero vector, and so either we're going to produce another separating axis
     ↪ perpendicular to the two vectors, or by finally determining that no separation can occur.
 *
 */
glm::tvec3<float> ca = cd.source - ab.source;
glm::tvec3<float> n = glm::cross(vectorAB, ca);
m = glm::cross(normalizeNotZeroVector(vectorAB), normalizeNotZeroVector(n));

// The computed cross product will be a valid separating axis if and only if the resulting vector is
     ↪ not a zero
// vector
return std::make_pair(m, !isZeroVector(m));
    } else {
return std::make_pair(m, false);
    }
}
```

We leave to the reader two simple overrides of the `operator<<` for both `glm::tvec3<float>` and `std ↪ ::pair<glm::tvec3<float>, bool>`. Then, we can create a simple test for checking that the previous implementation works as expected:

```
void vector_test_intersection() {
    // Getting the smallest number representable as float
    float e = std::numeric_limits<float>::epsilon();

    Vector ab{{1, 2, 3}, {0, 0, 0}};

    // Adding a minimum quantity that will be indistinguishable for non-robust algorithms
    Vector abDelta{{1 + e, 2 - e, 3 + e}, {0, 0, 0}};

    // false is the expected result: there should not be a separating axis (the two vectors are the
        ↪ same, and therefore they are parallel).
    std::cout << SepAxisCrossProd(ab, ab) << std::endl;

    // true is the expected result: there should be a separating axis given that the two vectors are
        ↪ indeed not the same, because they differ only by a small quantity (e)
    std::cout << SepAxisCrossProd(ab, abDelta) << std::endl;

    // false is the expected result: the algorithm is less robust, and then the test might fail.
    std::cout << SepAxisCrossProd(ab, abDelta, false) << std::endl;
}
```

Representing OBBs Given the previous lemma's requirements, we can now define the `struct OBB` that is going to held the information of the center point T (`center_point`), the rotation matrix decomposed in three `local_axes`, and the `halfwidth_sizes` vector.

```
// glm
#include <vec3.hpp>

// Vector
#include <numeric/structures/Vector.h>


struct OBB {
    Vector center_point;  // OBB Center point
    Vector local_axes[3]; // Local x-, y-, and z- axes
    Vector halfwidth_sizes; // Positive halfwidth extends of OBB along each axis

    /**
     * Setting up the OOB sitting at the origin, with the same axes as the cartesian ones, and
         ↪ specific half distances
     *
     * @param halfX
     * @param halfY
     * @param halfZ
     */
    OBB(float halfX, float halfY, float halfZ);

    /**
     * Setting up the OBB sitting at a given center, with the same axes as the cartesian ones, and
         ↪ specific half distances
     * @param c
     * @param e
     */
    OBB(const glm::tvec3<float>& c, const glm::tvec3<float>&e);

    /**
     * Setting up a custom OBB
     * @param c
     * @param x
     * @param e
     */
    OBB(const glm::tvec3<float>& c, const std::vector<Vector>& x, const glm::tvec3<float>&e);
};
```

The implementation of such struct is quite straightforward:

```
OBB::OBB(float halfX, float halfY, float halfZ) : center_point{0, 0, 0}, halfwidth_sizes{halfX, halfY,
    ↪ halfZ} {
    local_axes[0] = {1, 0, 0}; // Using the default copy constructor
    local_axes[1] = {0, 1, 0};
    local_axes[2] = {0, 0, 1};
}

OBB::OBB(const glm::tvec3<float> &c, const glm::tvec3<float> &e) : center_point{c}, halfwidth_sizes{e}
    ↪ {
    local_axes[0] = {1, 0, 0}; // Using the default copy constructor
    local_axes[1] = {0, 1, 0};
    local_axes[2] = {0, 0, 1};
}

OBB::OBB(const glm::tvec3<float> &c, const std::vector<Vector> &x, const glm::tvec3<float> &e) :
    ↪ center_point{c}, halfwidth_sizes{e} {
    local_axes[0] = x[0];
    local_axes[1] = x[1];
    local_axes[2] = x[2];
}
```

Implementing the Naïf Overlap Test First thing, we need to implement Equation 1 on page 15 for each possible axis L. Given that the dot product doesn't suffer from severe accuracy problems as the cross product, we might straightforwardly provide the following implementation:

```
/**
 * Checks whether the OBBs provided as parameters overlap over a given axis
 *
 * @param a First OBB
 * @param b Second OBB
 * @param L Axis over which we want to test the overlap
 * @return  Whether the overlap happens or not
 */
static inline bool chunkNotOverlapTest(OBB& a, OBB& b, const glm::tvec3<float>& L) {
    auto tDiff = a.center_point.asVector() - b.center_point.asVector();

    return std::fabs(glm::dot(tDiff, L)) /* s */ >
   (a.halfwidth_sizes[0] * std::fabs(glm::dot(a.local_axes[0].asVector(), L)) + //r_A
    a.halfwidth_sizes[1] * std::fabs(glm::dot(a.local_axes[1].asVector(), L)) +
    a.halfwidth_sizes[2] * std::fabs(glm::dot(a.local_axes[2].asVector(), L)))
   +
   (b.halfwidth_sizes[0] * std::fabs(glm::dot(b.local_axes[0].asVector(), L)) + //r_B
    b.halfwidth_sizes[1] * std::fabs(glm::dot(b.local_axes[1].asVector(), L)) +
    b.halfwidth_sizes[2] * std::fabs(glm::dot(b.local_axes[2].asVector(), L)));
}
```

Then, we can provide two distinct implementation of the Naïf Overlap Test: one where the robust cross product is never performed (`doRobust=false`), and another one where the robust cross product is always performed. We can now provide the following implementation:

```
/**
 * Using the naif overlap. This solution implements the mathematical constraint definition as it is,
     ↪ with no numerical
 * optimization (besides the cross product one)
 *
 * @param a    First OBB
 * @param b    Second OBB
 * @param doRobust    Whether we want to perform the robust cross product or not
 * @return     Returns whether the two OBBs overlap or not
 */
bool NaifOverlap(OBB &a, OBB &b, bool doRobust = true) {
    // Do return the first time that I see a separation: if the elements overlap, they shall overlap
        ↪ along all the axes. In the first six axes cases, we do not need to perform any cross
        ↪ product. This part is shared between the robust and the non robust implementation.
    if (chunkNotOverlapTest(a, b, a.local_axes[0].asVector())) return false;
    if (chunkNotOverlapTest(a, b, a.local_axes[1].asVector())) return false;
    if (chunkNotOverlapTest(a, b, a.local_axes[2].asVector())) return false;


    if (chunkNotOverlapTest(a, b, b.local_axes[0].asVector())) return false;
```

```
    if (chunkNotOverlapTest(a, b, b.local_axes[1].asVector())) return false;
    if (chunkNotOverlapTest(a, b, b.local_axes[2].asVector())) return false;

    // If we are not interested in the robust implementation of the cross product, then we can simply
        ↪ use the general purpose
    // cross product defined by the GLM library...
    if (!doRobust) {
     // Using the naive version, with no robust cross product computation
     if(chunkNotOverlapTest(a, b, glm::cross(a.local_axes[0].asVector(), b.local_axes[0].asVector()))
         ↪ return false;
     if(chunkNotOverlapTest(a, b, glm::cross(a.local_axes[0].asVector(), b.local_axes[1].asVector()))
         ↪ return false;
     if(chunkNotOverlapTest(a, b, glm::cross(a.local_axes[0].asVector(), b.local_axes[2].asVector()))
         ↪ return false;
     if(chunkNotOverlapTest(a, b, glm::cross(a.local_axes[1].asVector(), b.local_axes[0].asVector()))
         ↪ return false;
     if(chunkNotOverlapTest(a, b, glm::cross(a.local_axes[1].asVector(), b.local_axes[1].asVector()))
         ↪ return false;
     if(chunkNotOverlapTest(a, b, glm::cross(a.local_axes[1].asVector(), b.local_axes[2].asVector()))
         ↪ return false;
     if(chunkNotOverlapTest(a, b, glm::cross(a.local_axes[2].asVector(), b.local_axes[0].asVector()))
         ↪ return false;
     if(chunkNotOverlapTest(a, b, glm::cross(a.local_axes[2].asVector(), b.local_axes[1].asVector()))
         ↪ return false;
     if(chunkNotOverlapTest(a, b, glm::cross(a.local_axes[2].asVector(), b.local_axes[2].asVector()))
         ↪ return false;
    } else {
// ... otherwise, use the cross product enhanced for this specific problem, i.e. for computing a
    ↪ separating axis.
std::function<bool(OBB&, OBB&, int, int)> thereIsNoOverlap = [doRobust](OBB& a, OBB& b, int i, int j)
    ↪ {
    std::pair<glm::tvec3<float>, bool> result;
    // If there is a separating axis R_i^A × B_j^B
    if ((result = SepAxisCrossProd(a.local_axes[i], b.local_axes[j], doRobust)).second) {
// Check whether the elements do overlap or not
return (chunkNotOverlapTest(a, b, result.first));
    } else {
// otherwise, there is no separating axis because the cross product provides a nearly-zero vector:
    ↪ continue
return false;
    }
};

if (thereIsNoOverlap(a, b, 0, 0)) return false;
if (thereIsNoOverlap(a, b, 0, 1)) return false;
if (thereIsNoOverlap(a, b, 0, 2)) return false;
if (thereIsNoOverlap(a, b, 1, 0)) return false;
if (thereIsNoOverlap(a, b, 1, 1)) return false;
if (thereIsNoOverlap(a, b, 1, 2)) return false;
if (thereIsNoOverlap(a, b, 2, 0)) return false;
if (thereIsNoOverlap(a, b, 2, 1)) return false;
if (thereIsNoOverlap(a, b, 2, 2)) return false;
    }

    // If there is no separating axis, then the elements do overlap
    return true;
}
```

3.2 Optimized Test

As observed in the original work [3], the best way to get rid of the numerical cancellation error caused by the cross product is to get rid of the cross product. In order to make the resulting algorithm more robust, we might as well reduce the number of sums (Lemma 4 on page 8) and multiplications (Example 5 on page 8), as both operations might introduce increasing numeric errors. In order to do so, we might perform three optimizations in three distinct steps bearing in mind that that we're going to use exactly the axes provided in Definition 5 on page 15 and grouped in three different cases *(i)-(iii)* at page 15.

Candidate Axis Optimization Given that the `local_axes` are perpendicular by construction and that the dot product of perpendicular vectors is zero, then *(i)* for $L = R_i^A$ the test becomes:

$$|(T^A - T^B) \cdot R_i^A| > a_i + \sum_{0 \leq j \leq 2} b_j |R_j^B \cdot R_i^A| \tag{2}$$

Similarly, *(ii)* for $L = R_j^B$ the test becomes:

$$|(T^A - T^B) \cdot R_j^B| > \sum_{0 \leq i \leq 2} a_i |R_i^A \cdot R_j^B| + b_j \tag{3}$$

For the last remaining case, *(iii)* we have that all the candidates will be some $R_i^A \times R_j^B$ and, even in this case, we want to reduce the number of cross products as much as possible. After observing that the following equivalences hold:

$$\vec{u} \cdot (\vec{v} \times \vec{w}) = \vec{w} \cdot (\vec{u} \times \vec{v}) = \vec{v} \cdot (\vec{w} \times \vec{u})$$

we can remove some unwanted cross products by rewriting the expressions so that we obtain parallel cross products that can be then elided. E.g., $R_0^A \cdot (R_0^A \times R_1^B) = R_1^B \cdot (R_0^A \times R_0^A) = R_1^B$. Given that all the OBB components are orthogonal, e.g., $R_0^A \perp R_2^A$ and $R_1^A \perp R_2^A$, we might observe that the cross product of two vectors of the same OBB will inevitably provide the third direction vector, e.g., $R_0^A \times R_1^A = R_2^A$ by construction. Therefore, we will have that the only remaining cross product will be the one defining L:

$$|(T^A - T^B) \cdot (R_i^A \times R_j^B)| > \sum_{0 \leq i' \neq i \leq 2} a_{i'} |R_j^B \cdot R_{i'}^A| + \sum_{0 \leq j' \neq j \leq 2} b_{j'} |R_i^A \cdot R_{j'}^B| \tag{4}$$

OBB_A's Local Coordinates For this second step, we want to get rid of the last cross product and reduce the number of the dot products. We observe that if both boxes undergo the same rigid transformation (translation and rotation), then their overlap is unaltered. As a consequence, we might want to translate OBB_A to the origin of the Cartesian axes ($T^{A'} := (0,0,0) = \mathbf{0}$) and then rotate its axes so that R^A will represent the standard basis for $\mathbb{R}^3$ (i.e., the identity matrix $R^{A'} = I_3 = [\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2]$). Let us now focus on the required rigid transformation: with respect to the translation, we have that $T^{A'} = T^A - T^A$, so the required translation will be $-T^A$; with respect to the rotation, if R^A is an orthogonal matrix whose columns and rows are orthogonal unit vectors, then we can prove that $(R^A)^t = (R^A)^{-1}$, so that the linear transformation will become[7] $(R^A)^{-1}$.

Consequently, we have also to first translate OBB_B by $-T^A$ and then rotate it by $(R^A)^t$, thus obtaining the linear transformation $(R^A)^t(\vec{x} - T^A)$. This means that we can transform T^B as $T^{B'} := (R^A)^t(\vec{x} - T^A)$ and its rotation matrix $R^{B'}$ will become $(R^A)^t R^B$.

Now, we can continue to further optimize the former optimization steps: *(i)* for $L = R_i^A$, Equation 2 becomes:

$$|(\mathbf{0} - T^{B'}) \cdot \mathbf{e}_i| = \left|T^{B'}{}_i\right| > a_i + \sum_{0 \leq j \leq 2} b_j |R_j^{B'} \cdot \mathbf{e}_i| = a_i + \sum_{0 \leq j \leq 2} b_j |R_j^{B'}.i|$$

where $R_j^B.i$ means to access the i dimension of the R_j^B vector:

$$\left|T^{B'}{}_i\right| > a_i + \sum_{0 \leq j \leq 2} b_j |R_j^{B'}.i| \tag{5}$$

Similarly, *(ii)* for $L = R_j^B$ Equation 3 becomes:

$$|(\mathbf{0} - T^{B'}{}_i) \cdot R_j^{B'}| = |T^{B'}{}_i \cdot R_j^{B'}| > \sum_{0 \leq i \leq 2} a_j |\mathbf{e}_i \cdot R_j^{B'}| + b_j = \sum_{0 \leq i \leq 2} a_j |R_j^{B'}.i| + b_j$$

$$|T^{B'}{}_i \cdot R_j^{B'}| > \sum_{0 \leq i \leq 2} a_i |R_j^{B'}.i| + b_j \tag{6}$$

[7] $R^{A'} = I_3 = (R^A)^t R^A = (R^A)^{-1} R^A$

For the third case, *(iii)* Equation 4 becomes:

$$|(\mathbf{0} - T^B) \cdot (\mathbf{e}_i \times R_j^B)| > \sum_{0 \leq i' \neq i \leq 2} a_{i'} |R_j^B \cdot \mathbf{e}_{i'}| + \sum_{0 \leq j' \neq j \leq 2} b_{j'} |\mathbf{e}_i \cdot R_{j'}^B|$$
$$= \sum_{0 \leq i' \neq i \leq 2} a_{i'} |R_j^B.i'| + \sum_{0 \leq j' \neq j \leq 2} b_{j'} |R_{j'}^B.i|$$

In order to rewrite the left hand side of the inequality, we need to first prove the following lemma:

Lemma 6. *Given* $\vec{v} = (v_0, v_1, v_2)$*, then* $T^B \cdot (\mathbf{e}_i \times \vec{v}) = -T^B_{(i+1) \bmod 3} v_{(i+2) \bmod 3} + T^B_{(i+2) \bmod 3} v_{(i+1) \bmod 3}$

Proof. For $i = 0$, we have: $T^B \cdot (\mathbf{e}_0 \times \vec{v}) = T^B \cdot (0, -v_2, v_1) = -T_1^A v_2 + T_2^A v_1$.
For $i = 1$, we have: $T^B \cdot (\mathbf{e}_1 \times \vec{v}) = T^B \cdot (v_2, 0, -v_0) = -T_2^A v_0 + T_0^A v_2$.
For $i = 2$, we have: $T^B \cdot (\mathbf{e}_2 \times \vec{v}) = T^B \cdot (-v_1, v_0, 0) = -T_0^A v_1 + T_1^A v_0$. □

So, the last equation becomes:

$$| - T^B_{(i+1) \bmod 3} v_{(i+2) \bmod 3} + T^B_{(i+2) \bmod 3} v_{(i+1) \bmod 3}| > \sum_{0 \leq i' \neq i \leq 2} a_{i'} |R_j^B.i'| + \sum_{0 \leq j' \neq j \leq 2} b_{j'} |R_{j'}^B.i| \quad (7)$$

At this stage, we can see that *(i)* the first case has 4 absolute value operations, 3 multiplications and 3 sums/differences, *(ii)* the second case has 4 absolute values, 6 multiplications and 5 sums/differences, and *(iii)* in the third case we have 5 absolute values, 6 multiplications, and 4 sums/differences. As a gran total, we have 69 absolute values, 81 multiplications and 60 sums/differences. If we also consider the cost of the rigid coordinate transformation, when we still have 69 values, but have 117 multiplications and 87 sums/differences.

Last Optimization To save more operations, we can compute $|R^{B'}|$ in advance, thus reducing the number of absolute value computations to 24. Last, even in this case we need to take care about the near parallel axes cross product: the error introduced in the rotation matrix $|R^{B'}|$ has now a large influence. As the right-hand side of the test inequalities should be larger when two OBBs are overlapping, we can add a small `epsilon` quantity to $|R^{B'}|$, so that the right hand side is considerably increased [3].

After all of these optimization steps, we can provide the final implementation of such algorithm.

```
int RobustOBBOverlap(OBB &a, OBB &b, const float epsilon = 1e-6) {
    float ra, rb;
    glm::tmat3x3<float> R, AbsR;
    glm::tvec3<float> t;// Translation vector

    for (int i = 0; i<3; i++) {
for (int j = 0; j<3; j++) {
    // OBB_A's Local Coordinates
    float tmp = glm::dot(a.local_axes[i].asVector(), b.local_axes[j].asVector());
    R[i][j] = tmp;
    AbsR[i][j] = std::abs(tmp) + epsilon; // Last optimization
}
    }

    // OBB_A's Local Coordinates: computing the translation from b to a
    t = b.center_point.asVector() - a.center_point.asVector();
    // OBB_A's Local Coordinates: bringing translation into a's coordinate frame
    t.x = glm::dot(t, a.local_axes[0].asVector());
    t.y = glm::dot(t, a.local_axes[1].asVector());
    t.z = glm::dot(t, a.local_axes[2].asVector());


    for (int i = 0; i<3; i++) {
// Case (i)
ra = a.halfwidth_sizes[i];
rb = b.halfwidth_sizes[0] * AbsR[i][0] + b.halfwidth_sizes[1] * AbsR[i][1]
```

```
      + b.halfwidth_sizes[2] * AbsR[i][2];
if (std::fabs(t[i]) > ra + rb) return 0;

// Case (ii)
ra = a.halfwidth_sizes[0] * AbsR[0][i] + a.halfwidth_sizes[1] * AbsR[1][i]
      + a.halfwidth_sizes[2] * AbsR[2][i];
rb = b.halfwidth_sizes[i];
if (std::fabs(t[0] * R[0][i] + t[1] * R[1][i] + t[2] * R[2][i]) > ra + rb) return 0;
    }


    for (int i = 0; i<3; i++) {
for (int j = 0; j<3; j++) {
    // Case (iii)
    ra = a.halfwidth_sizes[(i + 1) % 3] * AbsR[(i + 2) % 3][j]
  + a.halfwidth_sizes[(i + 2) % 3] * AbsR[(i + 1) % 3][j];
    rb = b.halfwidth_sizes[(j + 1) % 3] * AbsR[i][(j + 2) % 3]
  + b.halfwidth_sizes[(j + 2) % 3] * AbsR[i][(j + 1) % 3];
    if (std::fabs(t[(i+2)%3] * R[(i+1)%3][j] - t[(i+1)%3] * R[(i+2)%3][j]) > ra + rb) return 0;
}
    }

    return 1;
}
```

Now, we can perform some tests over both provided implementations of the Gottshalk's Test. First, we can test that both implementations are able to tell that each OBB is always overlapping with itself, and that two non-intersectiing OBBs should fail the test.

```
// OOB with the same axes as the orthonormal basis, with half distance of 2 and center at the origin
    ↪ of the axes
OBB a{2, 2, 2};
// OOB with the same axes as the orthonormal basis, with half distance of 2 and centered in (10,10,10)
OBB b{{10, 10, 10}, {2, 2, 2}};

assert(NaifOverlap(a, a)); // Self containment == 1
assert(NaifOverlap(b, b)); // Self containment == 1
assert(!NaifOverlap(a, b)); // box b is centered in (10,10,10) and has a radius of 2, so it cannot
    ↪ overlap with b == 0
assert(!RobustOBBOverlap(a, b));
```

Similarly, both tests are able to correctly detect that box b is completely contained into a:

```
// OOB with the same axes as the orthonormal basis, with half distance of 2 and center at the origin
    ↪ of the axes
OBB a{2, 2, 2};
// OOB with the same axes as the orthonormal basis, with half distance of 1 and center at the origin
    ↪ of the axes
OBB b{1, 1, 1};

assert(NaifOverlap(a, a)); // Self containment == 1
assert(NaifOverlap(b, b)); // Self containment == 1
assert(NaifOverlap(a, b)); // a contains b, because a is bigger than b == 1
assert(RobustOBBOverlap(a, b));
```

Next, both algorithms seem to be robust even when there is a small overlap change:

```
// Getting the smallest number representable as float
float e = std::numeric_limits<float>::epsilon();
// A small change number to add some noise to the data
float small_change = 0.000001;

// OOB with the same axes as the orthonormal basis, with half distance of 1 and center at the origin
    ↪ of the axes
OBB a{1, 1, 1};
// OOB with the same axes as the orthonormal basis, with half distance of 1 and center at (2,2,2)
    ↪ minus a small
// distance. As a consequence, all the components shall overlap
OBB b{{2 - small_change, 2 - small_change, 2 - small_change}, {1, 1, 1}};

assert(NaifOverlap(a, a));// Self containment == 1
assert(NaifOverlap(b, b));// Self containment == 1
```

```
assert(NaifOverlap(a, b));// Partial overlap == 1
assert(NaifOverlap(a, b, false)); // Partial overlap == 1
assert(RobustOBBOverlap(a, b)); // If we use an epsilon which is bigger than e, 1e-6, then the test
    ↪ won't fail as expected.
assert(RobustOBBOverlap(a, b, e)); // Using a smaller epsilon for the test
```

Up until now, we only created AABBs, so the always used the identity matrix as a rotation matrix. Now, we shall test the algorithms over OBBs; we can start smoothly by adding one single rotation to OBB_B, so that make the two OBBs overlap:

```
std::vector<Vector> vectorByCols;
vectorByCols.emplace_back(-0.0641566f, 0, -0.99794f);
vectorByCols.emplace_back(0.0f, 1.0f, 0.0f);
vectorByCols.emplace_back(-0.99794f, 0, 0.0641566f);

OBB a{3.53553f, 1.76777f, 0.0f};
OBB b{{-0.147256f, 1.76777f, 1.80947f}, vectorByCols, {2.33155f, 0.565685f, 0.56452f}};

assert(RobustOBBOverlap(a, b));
assert(NaifOverlap(a, b));
assert(NaifOverlap(a, b, false));
```

Finally, we shall consider OBBs that which axes differ only by a small amout, let's say ϵ. Even if we're trying to run the Naïf Overlap test with the more numerical robust version of the cross product, the whole test will be not accurate, and only the more robust implementation will be successful:

```
std::vector<Vector> vectorByCols;
vectorByCols.emplace_back(1+e,-e,e);
vectorByCols.emplace_back(e,1-e,e);
vectorByCols.emplace_back(e,-e,1+e);

OBB a{1, 1, 1};
OBB b{{2 + e, 2 + e, 2 + e}, vectorByCols, {1, 1, 1}};

assert(NaifOverlap(a, a));// Self containment == 1
assert(NaifOverlap(b, b));// Self containment == 1

//Now, the naif algorithm fails in all the scenarios, even though we're trying to use the robust cross
    ↪ product
//implementation. As a consequence, it is better to always use more numerical stable algorithms for ad
    ↪ hoc
//solution instead of making each single component of a non-robust algorithm robust...
//assert(!NaifOverlap(a, b));
//assert(!NaifOverlap(a, b, false));

// Test passed for the robust version
assert(RobustOBBOverlap(a, b, e)); // Using a smaller epsilon for the test will make it pass
```

We can therefore conclude that optimizations require moderate mathematical knowledge and that optimization of specific non-numerically stable subproblems does not necessarily lead to a more robust algorithm.

4 Exercises

1. Convert -6.8_{10} to `IEEE` single float notation. Then, try to convert back the bit string to a base-ten decimal.
2. Introduce a novel `IntervalArithmetic` constructor, taking a real number α from a string and representing it as an interval $[fl_T(\alpha), fl(\alpha)]$.
3. Interval Arithmetic can be also used in more complicated use case scenarios for Collision Detection: investigate the ones described in [8] and referenced in [2].
4. Two dimensional objects can be also used: one of the two dimensions has halfsize of 0. You can now further reduce the number of computations.
5. Another application of Gottschalk's Algorithm is Ray Casting. Hint, you need to assume that one of the halfsizes is ∞.

References

[1] Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the ginac framework for symbolic computation within the C++ programming language. *J. Symb. Comput.*, 33(1):1–12, 2002.

[2] Christer Ericson. *Real-Time Collision Detection*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[3] Stefan Gottschalk. *Collision Queries using Oriented Bounding Boxes*. PhD thesis, Chapel Hill, 2000.

[4] Jason Gregory. *Game engine architecture*. CRC Press, 3 edition, 2019.

[5] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, USA, 2009.

[6] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, USA, 2001.

[7] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007.

[8] Stephane Redon, Abderrahmane Kheddar, and Sabine Coquillart. Fast continuous collision detection between rigid bodies. *Comput. Graph. Forum*, 21(3):279–287, 2002.

[9] John M. Snyder. Interval analysis for computer graphics. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1992, Chicago, IL, USA, July 27-31, 1992*, pages 121–130, 1992.

[10] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[11] Tong Xin, Pieran Marris, Ana Mihut, Gary Ushaw, and Graham Morgan. Accurate real-time complex cutting in finite element modeling. In *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2018) - Volume 1: GRAPP, Funchal, Madeira, Portugal, January 27-29, 2018*, pages 183–190, 2018.