
# Introduction to Numerical Computing

## Numerically Stable Collision Detection

Giacomo Bergami

2/10/2017

# Numerical Computing

# Objectives

- Understand the difference between continuous and discrete representation.
- Represent a decimal integer in binary format and vice versa.
- Recognizing float's sign, exponent and mantissa via bit fields.
- Understand how discrete representation generates errors via approximations.
- Determine which numeric function has the least total error.

# Introduction: Modelling

Modern methodologies for examining real world phenomena, for supporting production activities, mechanical design (CAD) to video games, includes the following approximation steps:

- Real World to Mathematical Model (*modelling*)
  - Everything is in "continuous" representation (e.g., $\forall n.\ \mathbb{R}^n$).
- Mathematical Model to Computational Representation (*numerical computing*)
  - The mathematical model is converted into a discrete representation (finite number of digits).

# Introduction: Towards Approximation

Given that the computer is a discrete machine, it can represent exactly models based on discrete sets of symbols.

- Discretization implies a reduction in precision.
- The solution obtained is calculated from a finite set of numbers through a finite number of machine operations and a finite set of possible numbers.

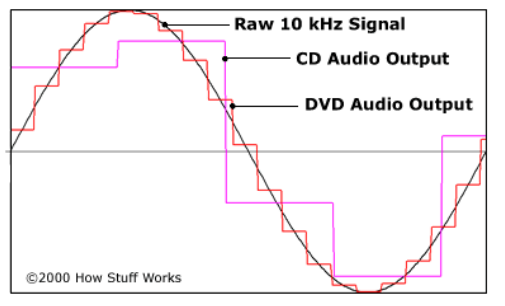


**Figure 1:** Analogue vs. Digital Soundwaves

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

# From Decimal To Binary and Viceversa

$$\mathbf{89}_{10} = 1011001_2$$

$$44 \cdot 2 + 1 = \mathbf{89}$$
$$22 \cdot 2 + 0 = 44$$
$$11 \cdot 2 + 0 = 22$$
$$5 \cdot 2 + 1 = 11$$
$$2 \cdot 2 + 1 = 5$$
$$1 \cdot 2 + 0 = 2$$
$$0 \cdot 2 + 1 = 1$$

$$1011001_2 = 1 \cdot 2^6 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 = \mathbf{89}_{10}$$

# Finite Numbers

The finite number system $F_{(\beta,t,\lambda,\omega)}$ The set of all the possible numbers representable in a basis $\beta \geq 2$ with $t$ significant cyphers and a interval bound over the exponent $p \in [\lambda, \omega]$:

$$F_{(\beta,t,\lambda,\omega)} = \{0\} \cup \left\{ \alpha \in \mathbb{R} \;\middle|\; \alpha = \pm \left( \sum_{i=0}^{t} \alpha_i \beta^{-i} \right) \beta^p, \lambda \leq p \leq \omega \right\}$$

This is also called *normalized representation*.

- The *machine epsilon* $\epsilon \geq \frac{1}{2}\beta^{1-t}$ is the distance between a number and the next value representable by a floating number.
  - $2\epsilon$ and $\epsilon$ will be respectively the truncating and rounding units.
- When $p \notin [\lambda, \omega]$, then we have representation errors:
  - *underflow*: $p < \lambda$
  - *overflow*: $p > \omega$

# IEEE Floating Point (1)

- Floating point represent real numbers using $n = 32$ bits. The bit representation determine the basis $\beta = 2$: then, such bits are distributed among the $t$ mantissa digits and the exponent $p$ that can have $\omega - \lambda + 1$ possible values. For binary representation, we assume that $\alpha_1$ is used to store the sign.
- Every operation $\oplus$ of floating point arithmetic approximating the exact $+$ operation is exact up to a relative error of size at most $\epsilon$: $x \oplus y = (x + y)(1 + \epsilon)$

# IEEE Floating Point (2)

See the `printing_float_representation()` within the source code for more concrete examples.

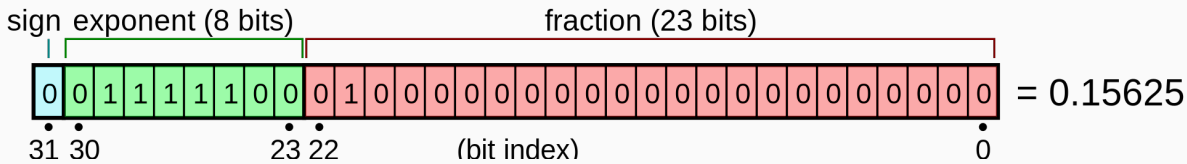


**Figure 2:** IEEE Single Precision Float Representation

# Error Propagation

- Numeric operation over finite numbers propagate errors, so that at each computation step, the final output will deteriorate. Given $u$ an architecture-specific rounding unit, we can prove that:
  - Multiplications and divisions are numerical stable: each operation provides an error of at most $3u$.
  - Sums and subtractions $x \pm y$ undergo *catastrophic cancellation* when $x \pm y \to 0$.
- Given two functions computing the same mathematical function, we prefer the one performing less mathematical operations.

# Error Propagation: Example

Let us suppose that we want to implement the pow operator, i.e.: $x^y$:

- An iterative approach multiplying $x$ $y-1$ times might seem a good idea (*iterative power*),
- but $x^y = e^{\log_e(x)\cdot y}$ is more numerically stable.
- Please use the default mathematical operations provided in the standard C/C++ library.
  - A possible implementation for double precision: `http://www.netlib.org/fdlibm/e_pow.c`

# Total Error

The total error is given by the discrepancy between the expected value returned by the perfect function and the one returned by the numeric function:

- Given a numeric function $\tilde{f}$ approximating a mathematical function $f$, then the total error of $\tilde{f}$ is defined as $\left|\frac{\tilde{f}(\tilde{n})-f(n)}{f(n)}\right|$, where $\tilde{n}$ is the numeric representation of $n$.

# Total Error: Example

Let us recall the definition of the Binet's Formula from the last tutorial: $F(n) = \frac{\sqrt{5}}{5}\left(\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$

- Implement a `binet_formula_error` using the *iterative power* function.
- Evaluate the total error between `binet_formula_error` and the value returned by the previously defined function.

We might observe from `total_error_plot()` that:

- we start to see a non-zero error from $n \geq 32$
- the error increases with $n$
  - $4.59 \cdot 10^{-7}$ for $n = 32$
  - $1.07 \cdot 10^{-6}$ for $n = 89$

# Gottschalk's Algorithm for OBB Overlap Tests

# Objectives

- Using the Separating Axis Theorem (SAT) for OBBs.
- Understand how to make specific operations more robust by implementing enhanced use-case specific functions.
- Understand how to make algorithms more robust by both reducing the number of operations and by removing (when possible) non-robust computations (e.g., cross products).

# Overlap Tests

- Linear Programming: test if at least one box corner is inside a box after representing each box via plane equations.
  - Requires to implement a too powerful framework for solution of linear programming problems.
- Distance Computation (GJK, LC): non-zero distance implies separation, while zero distance implies contact.
  - Too general for this simple problem.

# Separating Axis Theorem for AABB (1/3)

The Separating Axis Theorem holds for convex polygons (such as AABBs and OBBs).

- If we draw a straight line between two convex shapes (separating line), they do not overlap.
- This reduces to prove that two convex shapes do not overlap if there is an axis in which the two object's projections do not overlap.

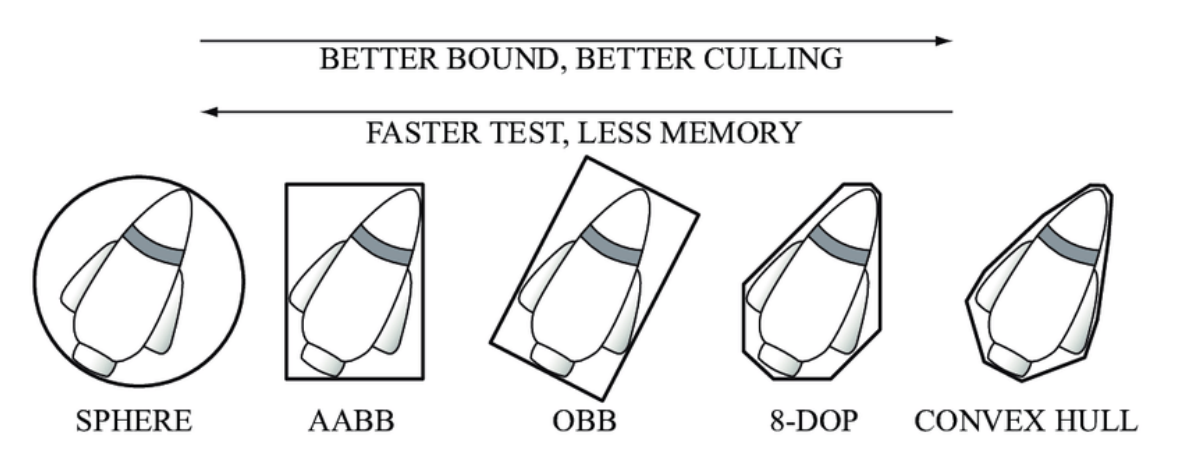


**Figure 3:** Example of some bounding boxes

# Separating Axis Theorem for AABB (2/3)

We can represent an AABB in the following ways:

- the minimum and maximum coordinate value
- the minimum coordinate value and the diameter extents from this corner
- the center coordinate and the halfwidth extents

# Separating Axis Theorem for AABB (2/3)

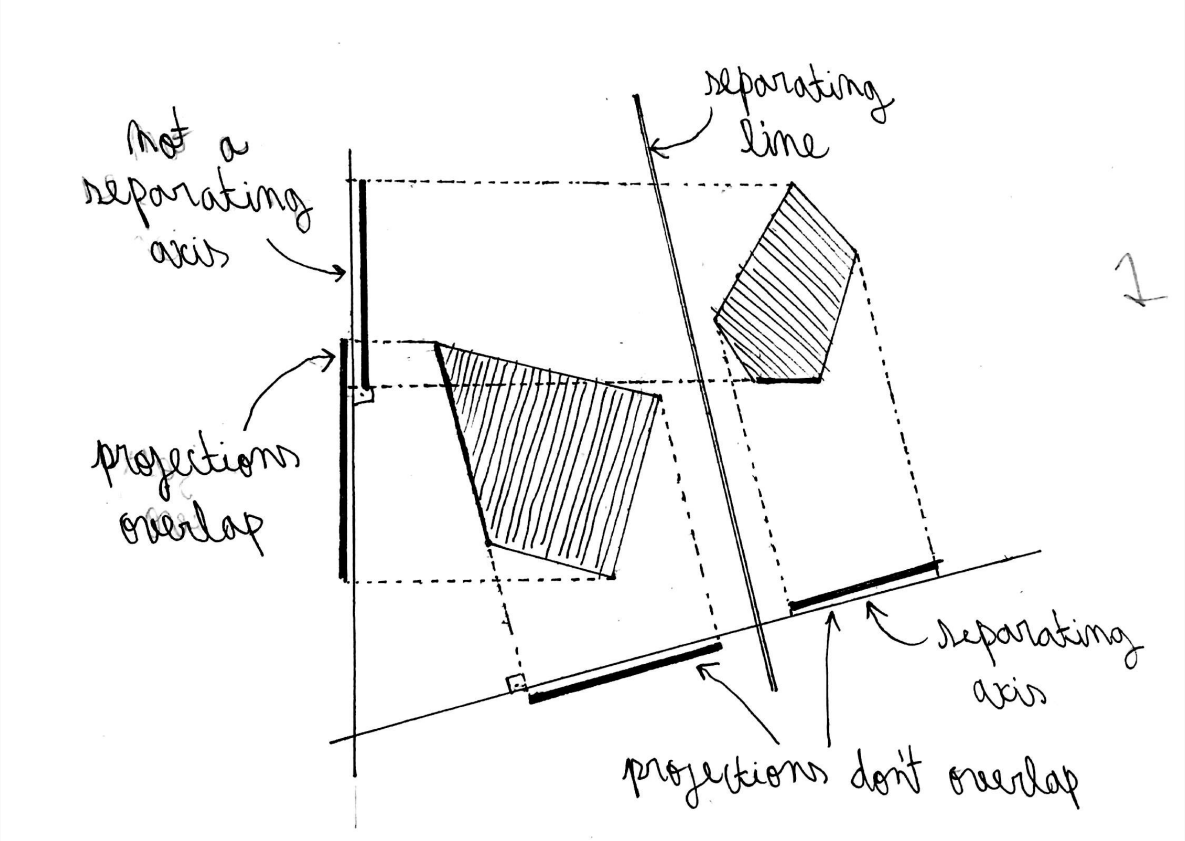


**Figure 4:** https://hackmd.io/@US4ofdv7Sq2GRdxti381_A/ryFmIZrsl

# Generalized Separating Axis Theorem (1/2)

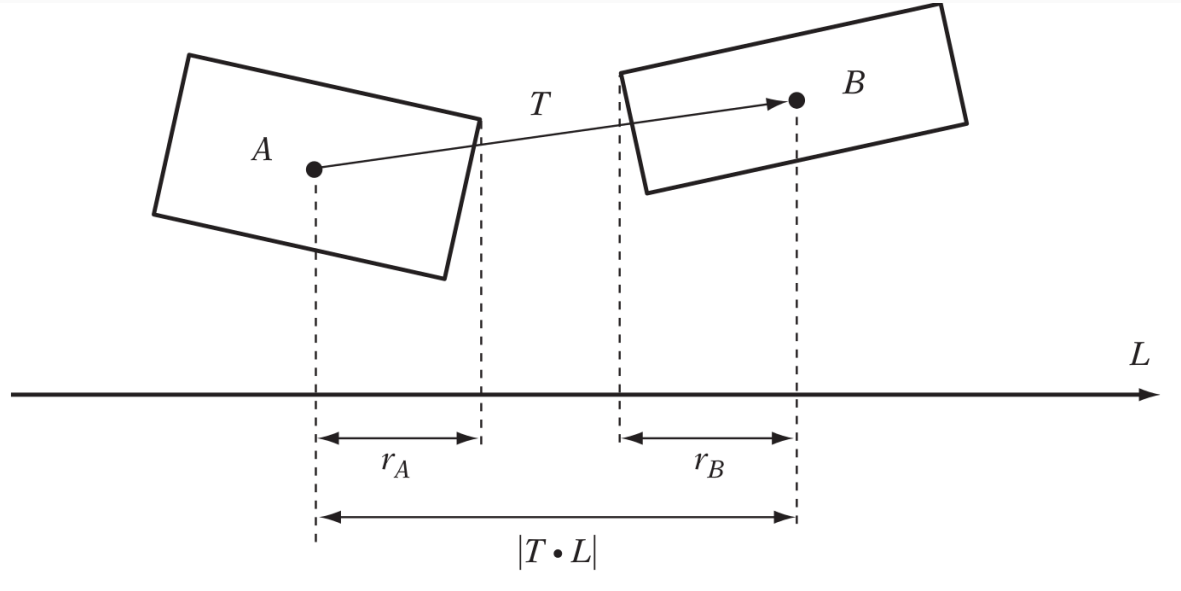


**Figure 5:** $L$ is the separation axis, represented as a vector. Let $T$ be the vector between the two OBBs centers and $L \cdot T$ its projection over the separation axis. The OBBs will be separated if the sum of the projected radii ($r_A$ and $r_B$) is less than $L \cdot T$.

$$|T \cdot L| > r_A + r_B$$

# Generalized Separating Axis Theorem (2/2)

- 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.
- The projection over the axis would require $O(V)$ time per axis. Using a vertex search data structure, then the test will take $O(V^2 \log V)$ per axis.
  - In his OBB Overlap Test PhD thesis, Stefan Gottschalk showed that we can reduce the number of comparisons to 15: we have only 3 distinct normals and 3 distinct face directions.

# OBB

An OBB $A$ is just an AABB with an arbitrary orientation. We can represent $A$ as:

- collection of 8 vertices
- collection of 6 planes
- a center point $T^A$, an orientation matrix $R^A$, three halfedge lengths $a_1, a_2, a_3$.

The last representation will provide a cheaper OBB-OBB intersection test, as it minimizes the number of operations required for the intersections (see previous slide).

# Generalized Separating Axis Theorem for OBBs

We can show that the previous statement reduces to the following computation:

$$|(T^A - T^B) \cdot n| / |n| > \frac{(a_1|R_1^A \cdot n| + a_2|R_2^A \cdot n| + a_3|R_3^A \cdot n|)}{|n|} + \frac{(b_1|R_1^B \cdot n| + b_2|R_2^B \cdot n| + b_3|R_3^B \cdot n|)}{|n|}$$

where $R_j^A$ is the $j$-th column of the matrix $R^A$, and each *candidate vector* $n$ is defined as follows:

- *case 1*: the 3 columns of $R^A$ (direction axes of $R^A$)
- *case 2*: the 3 columns of $R^B$ (direction axes of $R^B$)
- *case 3*: the 9 cross product of such columns.

# Problem/Solution 1

- When two direction axes are parallel, they provide a zero vector $n$ having $|n| = 0$.
- We want to avoid the division by zero, and therefore we can multiply all the sides by $|n|$. This is possible because:
  - If such axes are parallel, there will be no separation axis, and therefore the test might as well be skipped.
  - Zero norm axes will reduce to a zero length projection, leading to a $0 > 0$ test.

See `NaifOverlap` function with `doRobust=`**`false`**. See also `normalizeNotZeroVector`.

# Problem/Solution 2 (1/2)

Due to the catastrophic cancellation, we might have a zero vector when all its components are strictly smaller than $\epsilon$.

- Due to the catastrophic cancellation, nearly parallel vectors involved in the cross product might lead to a zero vector.
  - If they are nearly parallel, they will be in a same plane $P$: try to compute another axis perperndicular to one of the vector and in $P$.
- Vectors involved in the cross product with a large magnitude might affect the cross product computation.
  - Given that we only care for the vectors' directions, we can normalize all the vectors involved in the cross product.

See `NaifOverlap` function with `doRobust=`**`true`**.

# Problem/Solution 2 (2/2)

1: **function** SepAxisCrossProd($\vec{ab}, \vec{cb}$)
2: $\vec{v} = \vec{ab} \times \vec{cb}$
3: **if** $\vec{v} \approxeq \mathbf{0}$ **then**
4: **return** $\langle v, \texttt{true} \rangle$
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** $\vec{v} \approxeq \mathbf{0}$ **then return** $\langle v, \texttt{true} \rangle$
10: **elsereturn** $\langle \mathbf{0}, \texttt{false} \rangle$
11: **end if**
12: **end if**
13: **end function**

# Observation

- After solving the first problem, we might have 7 dot products, 7 absolute values, 1 vector subtraction, 6 vector-scalar multiplications and 4 scalar additions. This reduces to 39 multiplications, 21 additions/subtractions, and 7 absolute values for one axis. This reduces to 1005 arithmetic operations for all the 15 axes.
  - **Can we reduce the amount of arithmetic operations?**
- After solving the second problem, we can make an algorithm with the cross product problem more robust. Nevertheless, the algorithm would be even more robust if we might get rid of the cross product as a whole.
  - **Can we make the algorithm more robust?**
  - Hint, we need to perform additional mathematical operations and simplifications.

# Solution 3 (1/2)

1. If we expand the definition with the actual candidate vectors, we obtain the following simplifications:
   - For *case 1*, $r_a = a_j$ for each axis $R_j^A$.
   - For *case 2*, $r_b = b_j$ for each axis $R_j^B$.
   - For *case 3*, all the cross products on the right handside simplify into a dot product.
2. If we express $B$ in the coordinate system of $A$, we have that $R_A = I_3$, $T^A$ is the origin of the axes and $R^B$ is the composition of the two orientation matrices.
   - For *case 1*, the $R_A = I_3$ reduces the dot products into absolute values. We have no dot product.
   - For *case 2*, the $R_A = I_3$ reduces the dot products into absolute values. We have only one dot product per axis.
   - For *case 3*, the $R_A = I_3$ reduces the dot products into absolute values. We only have two dot products.

# Let's Code!

Further optimizations.

3. We can save more operations by computing $|R^B|$ in advance.
4. Adding a small number to $|R^B|$, i.e. $|R^B| + \varepsilon$, increases the right-handside computation when the vectors will be nearly zero.

After providing such operations, we have a function `RobustOBBOverlap` which is more robust than the optimized `NaifOverlap`.

- See the last test in `obb_overlap_testing`.

# Interval Arithmetic

# Objectives

- Understanding Interval Arithmetic
- Coping with rounding errors via Interval Arithmetics in some hard use case scenarios.

# Introduction

In some other scenarios, we cannot exploit mathematical tricks for generating a more robust version of the problem. In this scenario, we need to exploit another technique:

- Interval Arithmetic provides reliable results by bounding on rounding errors and measurement errors in mathematical computation. Some practical use cases are:
  - keeping track of and handling rounding errors directly during the calculation when no other optimization is possible
  - represent measurement uncertainties from physical and technical parameters
- In particular, we can express any real $x \in \mathbb{R}$ as a $[a, b]$ where $a, b \in F$ and $a \leq x \leq b$.

# Interval Arithmetic

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

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

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

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

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

See `examples_on_interval_arithmetic` for some implementation.

# Sphere-AABB Collision Detection

Given that we can describe a sphere of radius $r$ centered in $(c_x, c_y, c_z)$ as $S(p) = (p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 - r^2$ where all the points outside of the sphere $g$ have $S(g) > 0$, we can represent the same formula in interval arithmetic as follows:

$$\overline{S}(p) = (p_x - [c_x, c_x])^2 + (p_y - [c_y, c_y])^2 + (p_z - [c_z, c_z])^2 - [r, r]^2$$

So, we will have that $\min(\overline{S}(p)) > 0$ if $p$ lies outside the sphere.

Given the minimum and maximum coordinate representation of the AABB, we can represent it as three intervals

- See `sphere_AABB_intersection` for some implementation.

# Further Work

- 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.
- Another application of Gottschalk's Algorithm is Ray Casting. Hint, you need to assume that one of the halfsizes is $\infty$.
- Given the *machine epsilon* $\varepsilon$, we can express $x$ as a $x \pm \varepsilon$: change the current implementation of the `IntervalArithmetic` constructor over one single point accordingly.
- Investigate the uses of Inverval Arithmetic for other bounding boxes.