CSIT2013 IEEE proceedings, ISBN 978-1-4673-5825-5, pp.260-263, Jordan, 2013

# Projective Rational Arithmetic with Floating Point

Vaclav Skala
Department of Computer Science and Engineering
Faculty of Applied Sciences, University of West Bohemia
CZ 306 14 Plzen, Czech Republic
http://www.VaclavSkala.eu

*Abstract*—**Numerical computation is used nearly in all software packages. Today, computation is based on the floating point representation IEEE754 which offers single and double precision while quadruple or extended precision are supported very rarely in current programming languages. In this paper an alternative representation is presented which offers a higher range of digits of a fraction and a higher range of exponents. It is based on the projective extension of the Euclidean space. Due to this representation the division operation can be nearly eliminated, that leads to faster and more robust computation. The presented projective rational arithmetic is especially convenient for vector-vector architectures like GPU.**

***Keywords — numerical computation; floating point; IEEE754; precision of computation; linear algebra; computer graphics; scientific computation; projective geometry; GPU computation.***

## I. Introduction

Numerical computation is a kernel of computational packages in nearly all fields of applied computer science. Today's systems are processing larger and larger data sets, problems solved are more and more converging to ill conditioned numerical problems, computation is far from being precise and numerical representation is limited to integers and floating number representation with a limited precision using mostly IEEE754 standard. Numerical instability or imprecision caused several very expensive catastrophes, e.g. well known sinking of the Sleipner oil platform, Ariane spaceship failure [13] [15] etc.

In the "old ages" Algol68 programming language [22], [24] had the "**long**" construction, which enabled to extend numerical precision, e.g.

**long long real** pi = 3.14159 26535 89793 23846 26433
83279 50288 41971 69399 37510;

Algol68 had a significant influence to many programming languages including C, resp. C++ and it was used to write an operation system ICL VME, too. A good summary of the Algol 68 features is described in [24]. Several modifications, incomplete implementations and new implementations have been reported [23]. Unfortunately the "long" construction is not available in current programming languages and current floating representations are mostly limited to single or double precision; very rarely quad or extended precision is supported. Double precision seems to be sufficient, however especially in scientific computing, geometry computations etc. the precision is not acceptable and unreliability of computations is very often observed. One typical example is a matrix multiplication in the case of large matrices, when Strassen algorithm is used to speed-up the computation, but the error of computation is growing fast.

TABLE I. FLOATING POINTS BIT STRUCTURE

| | Name | Digits | E min | E max |
|---|---|---|---|---|
| B 16 | Half | 10+1 | −14 | 15 |
| B 32 | Single | 23+1 | −126 | 127 |
| B 64 | Double | 52+1 | −1022 | 1023 |
| B 128 | Quad | 112+1 | −16382 | 16383 |

The current IEEE 754 specification is highly optimized from the "technology" or "hardware" point of view. However there is a question whether this approach is viable in future. There are some implementations aiming exact computation, e.g. [4], but the computation is just slow. On the other hand in practice we usually accept a limited precision if the precision is high enough and the results have accuracy required.

In this paper an alternative approach using projective extension of the Euclidean representation will be presented and discussed.

## II. Alternative Representations

The problem with a precision can be solved by using exact arithmetic approach [4], but computational time is high in general. There are other alternative ways how to increase a precision of computation as in the majority of cases we do not need exact solution having very long fractional part. High attention given especially to:

- rational arithmetic based on fractions with integer numbers, i.e. a rational number is expressed as
$$P = \frac{i}{j}$$
where $i, j$ are integers
- continuous fractions, e.g.
$$\pi = \cfrac{4}{1 + \cfrac{1^2}{3 + \cfrac{2^2}{5 + \cfrac{3^2}{\dots}}}}$$

- logarithmic representation in which multiplication is becoming a summation, division is a subtraction etc. However summation itself is complicated requiring high memory capacity [1].

However there are several drawbacks if rational arithmetic with integers or continuous fractions are used:

- current processors are optimized to the IEEE754 floating point representation
- division operation is slow and leading to inaccuracy and to high energy consumption

The precision of computation can be estimated using interval arithmetic. If the x representation, resp. y is in the interval $x \in [a, b]$, resp. $y \in [c, d]$ then the precision of basic operations can be estimated as follows:

$$x + y \in [a + c, b + d]$$

$$x \in [a, b] \quad y \in [c, d]$$

$$x \times y \in [min(ac, ad, bc, bd), max(ac, ad, bc, bd)]$$

$$x / y = [min(a/c, a/d, b/c, b/d), max(a/c, a/d, b/c, b/d)]$$

$$\text{if } y \neq 0$$

It can be proved that the division operation is causing substantial inaccuracy of computation.

There is a question whether the projective extension of the Euclidean representation can bring advantages over the classical using the floating point representation – single or double precision.

## III. Projective representation

Projective extension of the Euclidean space is usually considered only as a “tool” useful for geometric transformations in computer graphics and computer vision.

A point $(X, Y) \in E^2$ is represented in projective space by homogeneous coordinates $\boldsymbol{x} = [x, y: w]^T$, where: $w$ is the homogeneous coordinate [2] [7] [10] [17] [18] [20]. The transformation from the projective space to the Euclidean space is given as

$$X = {}^{x}/_{w} \qquad Y = {}^{y}/_{w} \qquad \text{and} \qquad w \neq 0$$

In mathematics a different notation, more convenient, is used:

$$\boldsymbol{a} = [a_0: a_1, \dots, a_n]^T \qquad \boldsymbol{A} = [\frac{a_1}{a_0}, \dots, \frac{a_n}{a_0}]^T$$

It can be seen that the division operation can be “hidden” to the homogeneous coordinate $w$, resp. $a_0$. As the direct result of that is that the division operation is not needed, higher precision can be expected. However, it should be noted that problems with exponent overflow or underflow must be considered.

Using projective representation and principle of duality [3] [5] [6], several algorithms are simpler and easy to implement as well [8] [11] [16] [19] [21].

## IV. Basic operations using projective

Robustness of computation is a key issue in many sophisticated computational algorithms as sophisticated engineering problems solved today might be ill conditioned. Let us explore basic properties of the projective representation from a numerical precision and robustness point of view. However different data structures have to be considered, i.e.

- projective scalar $[a_0: a_1]$
- projective vector $[a_0: a_1, \dots, a_n]$ etc.

Fundamental operations with:

### *A. scalars*

- addition, resp. subtraction

$$[a_0: a_1] \pm [b_0: b_1] = [a_0 b_0: b_0 a_1 \pm a_0 b_1]$$

- multiplication

$$[a_0: a_1] * [b_0: b_1] = [a_0 b_0: a_1 b_1]$$

- division

$$[a_0: a_1] / [b_0: b_1] = [a_0 b_1: a_1 b_0]$$

### *B. vectors*

- addition resp. subtraction

$$[a_0: a_1, \dots, a_n] \pm [b_0: b_1, \dots, b_n] = [a_0 b_0: \{b_0(a_1, \dots, a_n) \pm a_0(b_1, \dots, b_n)\}]$$

- scalar multiplication (dot product)

$$[a_0: (a_1, \dots, a_n)] \cdot [b_0: (b_1, \dots, b_n)] = [a_0 b_0: \sum_{i=1}^{n} a_i b_i]$$

- vector multiplication (cross product)

$$[a_0: (a_1, \dots, a_3)] \times [b_0: (b_1, \dots, b_3)] = [a_0 b_0: \{(a_1, \dots, a_3) \times (b_1, \dots, b_3)\}]$$

Note that the projective vector is different from a vector which consists of projective scalars.

### *C. exponent normalization*

Exponents due to arithmetic operations tend to grow or become smaller. It means that the exponent overflow or underflow is to be check and exponents can be normalized. This is very simple operation as it means that the same value in the exponent is to be added or subtracted from both – numerator and denominator as well.

Extraction of an exponent for a single and double precision is defined as

EXP := (FP_value **land** MASK) **shr** m;

where: **land** is bitwise and operation, **shr** is shift right, MASK is the binary mask and *m* is the argument for the shift operation. In this way we can manipulate with the exponent, i.e. read or rewrite it. The sequence is quite simple; special cases are not considered. The corresponding values for different precision are defined in Tab.2.

TABLE II. HARDWARE CONSTATNTS

| Precision | MASK | m | Exp_Digits |
|---|---|---|---|
| Single | &7FC0 | 4 | 255 |
| Double | &7F80 | 7 | 2047 |

### *D. Comparison operation*

The comparison operation is a little bit tricky as the condition

$$a < b$$

i.e.

$$[a_0 : a_1] < [b_0 : b_1]$$

Projective scalars have to have homogeneous coordinate non-negative, i.e. $a_0 > 0$ and $b_0 > 0$

The condition is to be replaced as follows:

$$[b_0 a_1] < [a_0 b_1]$$

or the values must be multiplied by the sign of the homogeneous coordinate of the value. In practice this is not a complication as the condition is easy to implement in hardware for general case, if not limited to positive homogeneous value in general.

It should be noted that projective representation is used in computer graphics and computer vision naturally, not only for geometrical operations, but also for intersection computations etc.

## V. ADVANTAGES OF PROJECTIVE RATIONALS

There are some advantages and disadvantages of the proposed projective representation. Let us explore some of them for better understanding of a projective rational arithmetic application potential.

**Advantages**

- The mantissa is actually doubled due to the "hidden" division operation by the homogeneous value a there is a higher range of the fractional part
- The exponent range is higher. If a single precision is used, the range is $2^{-254}$ to $2^{254}$ , i.e.the range is actually $2^{508}$
- The division operation is eliminated by multiplication of a homogeneous value in which denominator is "hidden"
- Infinity can be handled properly, i.e. division by a value close or equal to zero does not cause "floating point overflow"
- If double precision for numerator and denominator is used is used, actually a quadruple extended precision is implemented; if quadruple precision is available we get more that 2 times better precision
- Simple implementation on vector-vector architectures, like GPU – available projective Library P-Lib [25]

**Disadvantages**

- Current hardware does not support projective rational floating point, but the additional computational cost of that is low, but should be considered
- Operations are approx. two times longer if not vector-vector architecture or SSE instructions are used
- Value of exponents have to be controlled – there is a possibility of exponent overflow or underflow, but easily solved by addition or subtraction to numerator and denominator in hardware.

  Normalization can be made in software without a significant slowdown of computations.
- There is a significant difference between vector of projective scalar values and projective vector, i.e.

TABLE III. REPRESENTATION OF VALUES

| Vector of projective scalars | Projective vector |
|---|---|
| $\left[\frac{a_1}{a_0^1}, \dots, \frac{a_n}{a_0^n}\right] \asymp [(a_0^1 : a_1), \dots, (a_0^n : a_n)]$ <br> $\asymp$ means equal projectively | $[a_0 : a_1, \dots, a_n]$ |

  and users have to be careful in the mathematical and expression formulations.
- In the case of iterative methods on the current CPU longer computation time is to be expected. The given approach is not convenient for application of iterative methods on CPU due to exponent values control in software

## VI. EXPERIMENTAL VERIFICATION

Let us consider following simple examples for demonstration of the proposed projective rational arithmetic in linear algebra.

**Gauss elimination**

Gaussian elimination method is well known for solving a system of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$ where:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

It generally produces an upper triangular matrix, *row echolon form*, and then solves the unknown $\boldsymbol{x}$ in the backward run.

The structure of the Gaussian elimination method is

```
for k := 1 to n
    for i := k+1 to m
        for j := k+1 to m
            $a_{ij} := a_{ij} - \frac{a_{ik} a_{kj}}{a_{kk}}$
```

The expression for $a_{ij}$ can be rewritten as

$$a_{ij} = \frac{a_{ij}a_{kk} - a_{ik}a_{kj}}{a_{kk}}$$

As the value of $a_{kk}$ can be very small, i.e. $a_{kk} \rightarrow 0$, and division by a denominator could cause significant inaccuracy or floating point overflow, exchange of rows is made in practice. If the projective notation is used

$$a_{ij} = [a_{kk} \colon a_{ij}a_{kk} - a_{ik}a_{kj}]$$

where: $a_{kk}$ is the homogeneous part of the expression. If $a = [a_0 \colon a_1]$ and $\overline{a} = [a_1 \colon a_0]$ is the reciprocal value on $a$, then we can write

$$a_{ij} = [\overline{a_{kk}} * (a_{ij}a_{kk} - a_{ik}a_{kj})]$$

It should be noted that a reciprocal value $\overline{a} = [a_1 \colon a_0]$ is actually a swap of $a_0$ and $a_1$ values. The scalar value 0 represented in the projective notation is $a = [1 \colon 0]$, i.e. the homogeneous value is non-zero, usually 1. It means that no division operation is needed, however it is still "hidden" in the homogeneous coordinate and pivoting has to be used.

As the solution of a system linear equations $\boldsymbol{Ax} = \boldsymbol{b}$ is equivalent to extended cross-product [9] [12] [14] we can write

$$\boldsymbol{\Omega}\, \boldsymbol{\xi} = \boldsymbol{0}$$

where

$$\boldsymbol{\Omega} = [-\boldsymbol{b}|\boldsymbol{A}] = \left[\begin{array}{c|ccc} -b_1 & a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ -b_n & a_{n1} & \cdots & a_{nn} \end{array}\right] = \begin{bmatrix} \boldsymbol{\omega}_1 \\ \vdots \\ \boldsymbol{\omega}_n \end{bmatrix}$$

and

$$\boldsymbol{x} = \boldsymbol{\omega}_1 \times \boldsymbol{\omega}_2 \times \ldots \times \boldsymbol{\omega}_n$$

where $\boldsymbol{x}$ is the solution of the linear system of equations in the projective form, i.e. $\boldsymbol{x} = [x_0 \colon x_1, \ldots, x_n]$.

It can be seen that no division is required unless we need to express the computed value in the Euclidean space.

As direct consequence of the equivalence we can easily solved many computational problems without the division operation.

The projective rational arithmetic with floating point has been verified experimentally for stability and precision of computation and inversion of the Hilbert matrix was used, which converge to a singular matrix with the growing size. The experiments proved the expected properties of the proposed approach, details can be found in [12]. [25].

## VII. Conclusion

Projective rational arithmetic with floating point was described and fundamental arithmetic operations were described. The projective representation using homogeneous coordinates is used in computer graphics and computer vision and its application enabled to solve many problems in more effective way. As it was shown the projective representation is convenient for general numerical computation as well as it has several advantages of the standard single or double floating point representation. From the precision point of view, it offers higher range of exponents and also significantly wider range for a fraction representation.

The presented approach is convenient for vector-vector hardware architectures including GPU. If used on CPU with SSE instructions it is slightly slower than the computation with the Euclidean notation, but offers higher precision natively. If double representation is used it offers more than quadruple or extended representation. It means that in several engineering computations it can be used to significantly increase precision of computation. Current C# implementation of the P-Lib library [25] uses current GPU hardware and it was intended for geometrical applications in $E^3$. In the future work the P-Lib library will be modified for numerical computations in general and optimized.

## Acknowledgment

The author would like to thank to anonymous reviewers for their critical comments and hints that improved this paper significantly. Thanks also belong to students and colleagues at the University of West Bohemia, Plzen and VSB-Technical University, Ostrava for their suggestions. The project was supported by the Ministry of Education of the Czech Republic, project No. LH12181.

## References

[1] Arnold,M.G: "A VLIW Architecture for Logarithmic Arithmetic", Proceedings of the Euromicro Symposium on Digital System Design (DSD'03), IEEE, 2003

[2] Bloomenthal,J., Rokne,J.: "Homogeneous Coordinates", The Visual Computer, Vol.11,No.1, pp.15-26, 1994.

[3] Coxeter,H.S.M.: "Introduction to Geometry", John Wiley, 1969.

[4] Fogel,E., Halperin,D., Wein,R.: "CGAL Arrangements and Their Applicaqtions – A Step by Step", Springer Verlag, 2012

[5] Hartley,R, Zisserman,A.: "MultiView Geometry in Computer Vision", Cambridge Univ. Press, 2000.

[6] Johnson,M.: "Proof by Duality: or the Discovery of 'New' Theorems", Mathematics Today, December 1996.

[7] Skala,V.: "A New Approach to Line and Line Segment Clipping in Homogeneous Coordinates", The Visual Computer, Vol.21, No.11, pp.905 914, Springer Verlag, 2005

[8] Skala,V.: "Length, Area and Volume Computation in Homogeneous Coordinates", International Journal of Image and Graphics, Vol.6., No.4, pp.625-639, 2006

[9] Skala,V.: "Barycentric Coordinates Computation in Homogeneous Coordinates", Computers & Graphics, Elsevier, Vol. 32, No.1, pp.120-127, 2008

[10] Skala,V.: "Intersection Computation in Projective Space using Homogeneous Coordinates", Int.Journal on Image and Graphics, Vol.8, No.4, pp.615-628, 2008

[11] Skala,V.: "Geometry, Duality and Robust Computing in Engineering", WSEAS Trans.on Computers, Vol.11, No.9, ISSN 1109-2742, E-ISSN 2224-2872, pp.275-291, 2012

[12] Skala,V: "Duality and Intersection Computation in Projective Space with GPU Support", WSEAS Trans.on Mathematics, Vol.9.No.6.pp.407-416, 2010

[13] Skala,V.: "Duality and Intersection Computation in Projective Space with GPU support", Applied Mathematics, Simulation and Modeling - ASM 2010 conference, NAUN, Corfu, Greece, pp.66-71, 2010

[14] Skala,V.: "Duality and Robust Computation", ICCOMP12, 16th WSEAS Int.Conf. on Computers ICCOMP12, Kos, Greece, pp.172-117, 2012

[15] Skala,V.: "Projective Geometry and Duality for Graphics, Games and Visualization", Course SIGGRAPH Asia 2012, Singapore, ISBN 978-1-4503-1757-3, 2012

[16] Skala,V.: "S-Clip E2: A New Concept of Clipping Algorithms", SIGGRAPH Asia Posters, accepted for publication, 2012

[17] Skala,V., Kaiser,J., Ondracka,V.: "Library for Computation in the Projective Space", 6th Int.Conf. Aplimat, Bratislava, pp. 125-130, 2007

[18] Stolfi,J: "Oriented Projective Geometry", Academic Press, 2001.

[19] Thomas,F., Torras,C.: "A Projective invariant intersection test for polyhedral", The Visual Computer, Vol.18, No.1, pp.405-414, 2002

[20] Yamaguchi,F.: "Computer-Aided Geometric Design – A Totally Four Dimensional Approach", Springer Verlag, 1996

[21] Yamaguchi,F., Niizeki,M.: "Some Basic Geometric Test Conditions in Terms of Plücker Coordinates and Plücker Coefficients", The Visual Computer, Vol.13, pp.29-41, 1997

*WEB references*

[22] Algol 68 - http://en.wikipedia.org/wiki/ALGOL_68 #Operating_Systems_written_in_ALGOL_68

[23] Algol 68- download, http://algol68.sourceforge.net/

[24] Algol68 – summary, http://en.wikipedia.org/wiki/ALGOL_68

[25] C# Library for computation in the projective space P-Lib, download http://www.kiv.zcu.cz/vyzkum/software/index.html

## APPENDIX A

The cross product in 4D defined as

$$\boldsymbol{x}_1 \times \boldsymbol{x}_2 \times \boldsymbol{x}_3 = det \begin{bmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} & l \\ x_1 & y_1 & z_1 & w_1 \\ x_2 & y_2 & z_2 & w_2 \\ x_3 & y_3 & z_3 & w_3 \end{bmatrix}$$

can be implemented in Cg/HLSL on GPU as follows:

```
float4 cross_4D(float4 x1, float4 x2, float4 x3)
{
        float4 a;
        a.x=dot(x1.yzw, cross(x2.yzw, x3.yzw));
        a.y=-dot(x1.xzw, cross(x2.xzw, x3.xzw));
        // or a.y=dot(x1.xzw, cross(x3.xzw, x2.xzw));
        a.z=dot(x1.xyw, cross(x2.xyw, x3.xyw));
        a.w=-dot(x1.xyz, cross(x2.xyz, x3.xyz));
        // or a.w=dot(x1.xyz, cross(x3.xyz, x2.xyz));

        return a;
}
```

or more compactly

```
float4 cross_4D(float4 x1, float4 x2, float4 x3)
{
        return ( dot(x1.yzw, cross(x2.yzw, x3.yzw)),
        -dot(x1.xzw, cross(x2.xzw, x3.xzw)),
        dot(x1.xyw, cross(x2.xyw, x3.xyw)),
        -dot(x1.xyz, cross(x2.xyz, x3.xyz)) );
}
```