INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Euclidean Division Implemented with a Floating-Point Division and a Floor

Vincent Lefèvre

N° 5604

Juin 2005

Thème SYM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5604--FR+ENG

The Euclidean Division Implemented with a Floating-Point Division and a Floor

Vincent Lefèvre

Thème SYM — Systèmes symboliques
Projet SPACES

Rapport de recherche n° 5604 — Juin 2005 — 16 pages

Abstract: We study conditions under which the Euclidean division can be implemented using a floating-point division followed by a floor function. We show that under reasonable assumptions, the rounding downward mode can always be used, and the rounding to nearest mode can be used in most practical cases. These results may be useful for any language, but there is a particular benefit for languages, like ECMAScript, that do not have an integer division and that always round to nearest. We also show that an intermediate extended precision can introduce errors and give a condition under which an extended precision has no effect on the results.

Key-words: floating-point arithmetic, Euclidean division, integer division, correct rounding, programming languages, ECMAScript, XPath

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

La division euclidienne implémentée avec une division flottante et une partie entière

Résumé : Nous étudions des conditions sous lesquelles la division euclidienne peut être implémentée à l'aide d'une division flottante suivie d'une fonction floor. Nous montrons qu'avec des hypothèses raisonnables, le mode d'arrondi vers le bas peut toujours être utilisé, et que le mode d'arrondi au plus près peut être utilisé dans la plupart des cas usuels. Ces résultats peuvent être utiles pour n'importe quel langage, mais surtout pour les langages, comme ECMAScript, qui n'ont pas de division entière et qui arrondissent toujours au plus près. Nous montrons également qu'une précision étendue intermédiaire peut introduire des erreurs et donnons une condition sous laquelle une précision étendue n'a aucun effet sur les résultats.

Mots-clés : arithmétique virgule flottante, division euclidienne, division entière, arrondi correct, langages de programmation, ECMAScript, XPath

1 Introduction

The *Euclidean division* of a nonnegative real number x by a positive real number y is mathematically defined by $\lfloor x/y \rfloor$, i.e. the largest integer smaller or equal to x/y. When both x and y are of integer type, the Euclidean division, also called the *integer division*, is well supported by most programming languages; its implementation is straightforward, without any rounding problem such as those encountered with floating-point arithmetic. The case where x and y are floating-point variables is less common, but still interesting. First, floating-point variables may be used to represent integers exactly, in particular when floating-point arithmetic allows to give more precision than integer arithmetic (for instance, 53 bits on processors with 32-bit integer registers) or when it leads to a faster implementation. But there are also cases where x and y are not necessarily integers, as in the algorithm that computes a lower bound between a segment and $\mathbb{Z}^2$ [6, 7], though a scaling can be performed to work on integers. Moreover the ECMAScript language does not have an integer division [1, Section 11.5.2] (mainly because it does not have an integer type), so that the programmer may wish to perform it using a division and the `floor` function from the Math object[1]; other languages, like the ones based on XPath 1.0 [8], may need the same trick.

A solution to implement a Euclidean division in floating-point arithmetic using basic mathematical functions (available in programming languages) is to perform a division x/y followed by the *floor* function (or equivalently, the *trunc* function, as x/y is here nonnegative). But as the result of the division is rounded, one does not necessarily obtain the wanted result $\lfloor x/y \rfloor$. This paper deals with conditions under which the obtained result is guaranteed to be $\lfloor x/y \rfloor$.

In Section 2, we give some definitions and requirements on the floating-point system. Section 3 explores the main cases that can occur, in short, what happens depending on how the results are rounded. Amongst these cases, one is difficult and the corresponding property is proved in Section 4. And as some platforms (e.g., Linux/x86) may compute intermediate results with an extended precision, in Section 5, we analyze the effect of such an intermediate precision on the final result.

[1]Indeed, this is the solution used in practice and recommended by many programmers, but without any justification.

2 The Floating-Point System

We consider a floating-point system in base 2 (the main notions on floating-point arithmetic can be found in [3]). Let n denote the mantissa size ($n \geq 2$). We assume that the division is correctly rounded in the target precision. An IEEE-754 [5] compliant system satisfies these properties as long as the system does not use extended precision for intermediate results.

The exponent of a nonzero floating-point number x is defined as the exponent of its most significant bit: $e_x = \lfloor \log_2(|x|) \rfloor$. But equivalent definitions up to a given constant[2] are also valid in the following.

If the system has bounded exponents, we assume that the system and the manipulated values satisfy the following properties:

- the maximum exponent is large enough so that computing x/y does not generate an overflow, even after rounding;
- the minimum exponent allows to represent $1 - 2^{-n}$ exactly. With this assumption, the floating-point system may be either with or without subnormals. Note that since $1 - 2^{-n}$ has exactly n bits 1, any floating-point number greater or equal to $1/2$ is a normal number. As a consequence, any integer up to 2^n is either exactly representable or above the maximum representable number.

3 The Different Cases

Let $k = \lfloor x/y \rfloor$, where the division and the floor function are the mathematical functions, i.e. without any rounding. If $x/y > 2^n$, then k is not necessarily exactly representable in the floating-point system. So, in the following, we assume that $x/y \leq 2^n$. If this inequality does not hold (which should be very rare in practice), then the problem depends on how the user wants k to be rounded; this very particular case is out of the scope of this paper.

We are interested in the conditions under which Algorithm 1 returns a value equal to k, i.e. $\lfloor \diamond(x/y) \rfloor = k$, where $\diamond$ denotes the active rounding mode.

[2] Other common definitions choose the value e such that the floating-point number is written $m \times 2^e$ with $1/2 \leq m < 1$ or with $2^{n-1} \leq m < 2^n$.

Algorithm 1 — Floating-point division and a floor.

```
d = x / y;
f = floor(d);
return f;
```

In the following sections, let us consider the various possible rounding modes: rounding downward, rounding to nearest, rounding upward.

3.1 Rounding Downward

This rounding mode is defined on the nonnegative real numbers by:

$\diamond(u)$ is the largest floating-point number less or equal to u.

As u is nonnegative, this rounding mode covers the conventional rounding toward $-\infty$ and toward 0 modes.

Here we have $\diamond(x/y) \leq x/y$, and since $k \leq x/y$ and is exactly representable, then $k \leq \diamond(x/y)$. Moreover $k + 1 > x/y \geq \diamond(x/y)$. Therefore $\lfloor \diamond(x/y) \rfloor = k$.

Thus, in the rounding downward mode, the wanted equality $\lfloor \diamond(x/y) \rfloor = \lfloor x/y \rfloor$ is always satisfied.

So, we could perform these operations always in this rounding mode. But if the active rounding mode is dynamical, like on most current processors, then restricting to this rounding mode could lead to a loss of performance. This is why we also need to consider the other rounding modes.

3.2 Rounding to Nearest

This rounding mode is defined by:

$\diamond(u)$ is the floating-point number that is the nearest to u.

The general rule for halfway cases is to round to the floating-point number that has an even mantissa, but we do not enforce any rule in the following for these cases. This would not be necessary, as such cases can never occur with a division (except in the subnormal zone, which can be ruled out as the floor function would give the result 0 in any case). Indeed, if $x/y = u$ where u is greater than the smallest normal positive number and is the middle of two consecutive representable numbers, i.e. if u has an $n+1$-bit odd mantissa, then the $y.u$ cannot be representable with an n-bit mantissa.

Taking this rounding mode into account is important since this is the default rounding mode as required by the IEEE-754 standard [5, Section 4.1]. Moreover, in some languages (Java [4], ECMAScript [1], XPath 1.0 [8], etc.), this is the only rounding mode.

Contrary to the rounding downward mode, one can find examples for which $\lfloor \diamond(x/y) \rfloor \neq k$ in the rounding to nearest mode. If $n \geq 2$, then $x = 3 \times 2^{n-1} + 2$ and $y = 3$ yield an example for which $\lfloor \diamond(x/y) \rfloor \neq k$. Indeed, $k = 2^{n-1}$, whereas $\diamond(x/y) = 2^{n-1} + 1$ (since $x/y = 2^{n-1} + 2/3$ and $2/3 > 1/2$).

However, if both x and y are n-bit integers, possibly scaled by a same power of two, then one can prove that $\lfloor \diamond(x/y) \rfloor = k$. This condition is satisfied if floating-point arithmetic is used as an implementation of a fixed-point arithmetic; in particular, it is satisfied by the values that occur in [6, 7]. One can even find a weaker condition on x and y, and under this condition, the assumption $x/y \leq 2^n$ is no longer necessary:

Theorem 1 *Consider a floating-point system in base 2, with or without subnormals, such that $1/2$ is a normal representable number. If u is a real number, let $\diamond(u)$ denote the floating-point number that is the nearest to u (in halfway cases, any of the enclosing floating-point numbers can be chosen). Let x and y be floating-point numbers such that $x \geq 0$, $y > 0$ and $x - y$ is exactly representable in the floating-point system. Assume that x/y does not overflow. Then $\lfloor \diamond(x/y) \rfloor = \lfloor x/y \rfloor$, and this value is exactly representable.*

This theorem is proved in Section 4. Note that its conditions are rather strong. Indeed, if we slightly weaken the condition on $x - y$, e.g. by just requiring $x - y$ to have $n + 1$ bits (for the mantissa), then one can find counter-examples. For instance, if $n \geq 2$, $y = 2^n - 1$ and $x = 3y - 1 = 3 \times 2^n - 4$, then all the conditions would be satisfied: y is representable, since it is an n-bit integer; x is representable, since it is an integer less than 2^{n+2} and a multiple of 4 (its last two bits are zeros); $x - y = 2y - 1 = 2^{n+1} - 3$ is representable on $n+1$ bits. But $x/y = 3 - 1/y > 3 - 2^{1-n}$, so that $\diamond(x/y) = 3$, and $\lfloor x/y \rfloor = 2 \neq \lfloor \diamond(x/y) \rfloor$.

3.3 Rounding Upward

This rounding mode is defined on the nonnegative real numbers by:

$\diamond(u)$ is the smallest floating-point number greater or equal to u.

As u is nonnegative, this rounding mode covers the conventional rounding toward $+\infty$ and away from 0 modes.

Contrary to the rounding to nearest mode, one can find examples for which both x and y are n-bit integers (i.e. $x, y < 2^n$) and $\lfloor \diamond(x/y) \rfloor \neq k$ in such a rounding mode. For instance, if $n = 4$, $x = 14$ and $y = 3$, then $14/3 = 100.101010\ldots$ in binary, so that $\diamond(14/3) = 101.0_2 = 5$. Therefore $\lfloor \diamond(14/3) \rfloor = 5$, whereas $k = \lfloor 14/3 \rfloor = 4$.

More generally, if $n \geq 4$, then the n-bit integers $x = 3 \times 2^{n-2} + 2$ and $y = 3$ yield an example for which $\lfloor \diamond(x/y) \rfloor \neq k$.

To guarantee a correct result, stronger conditions on x and y are necessary. For instance, if rounding to nearest is used in a system with a mantissa size of $n-1$ bits, then the result is correct in our n-bit system with upward rounding. So, one can reuse the conditions of Section 3.2 with a precision of $n - 1$ bits. In particular, if x and y are positive integers less than 2^{n-1}, then $\lfloor \diamond(x/y) \rfloor = \lfloor x/y \rfloor$ in any rounding mode.

4 Proof of Theorem 1

First, let us prove that $\lfloor x/y \rfloor$ is representable. If $x = 0$, then $\lfloor x/y \rfloor = 0$, which is representable. Otherwise we can define the *unit in the last place* of x, denoted $\operatorname{ulp}(x)$, as the weight of the n-th bit of the mantissa of x. We distinguish two cases:

- If $y \geq \operatorname{ulp}(x)$, then $x/y \leq x/\operatorname{ulp}(x) < 2^n$, and $\lfloor x/y \rfloor$ is an n-bit integer, which is representable.
- If $y < \operatorname{ulp}(x)$: Since $x - y$ is representable, the difference between the exponent of its most significant bit and the exponent of its least significant nonzero bit is at most $n - 1$. Therefore its exponent is necessarily less than the exponent of x; and since $n \geq 2$, we have $y < \operatorname{ulp}(x) \leq x/2^{n-1} \leq x/2$, thus $x - y > x/2$, so that the exponent of $x - y$ is the exponent of x minus 1. As a consequence,

x is a power of two and $y = \mathrm{ulp}(x)/2$. It follows that $\lfloor x/y \rfloor = 2^n$, which is representable.

We now seek to prove that $\lfloor \diamond(x/y) \rfloor = \lfloor x/y \rfloor$.

4.1 Case $x < y$

In this case, $\lfloor x/y \rfloor = 0$. So, we seek to prove that $\diamond(x/y) < 1$.

Let y^- denote the largest floating-point number less than y. Then we have: $x/y \leq y^-/y$; therefore $\diamond(x/y) \leq \diamond(y^-/y)$. This means that it is sufficient to do the proof for $x = y^-$.

The value of $\diamond(y^-/y)$ does not change if y is scaled by a power of two. For the sake of simplicity, we can assume that $1/2 < y \leq 1$. Then $y^- = y - 2^{-n}$, and $y^-/y = 1 - 2^{-n}/y \leq 1 - 2^{-n}$, which is exactly representable (as assumed in Section 2). Therefore $\diamond(y^-/y) \leq 1 - 2^{-n} < 1$.

As a summary, $\diamond(x/y) \leq \diamond(y^-/y) < 1$, hence $\lfloor \diamond(x/y) \rfloor = 0 = \lfloor x/y \rfloor$.

4.2 Case $x = y$

This case is trivial: we have $\diamond(x/y) = x/y = 1$, hence $\lfloor \diamond(x/y) \rfloor = \lfloor x/y \rfloor$.

4.3 Case $x > y$

If $\diamond(x/y) \leq x/y$, then $\lfloor \diamond(x/y) \rfloor = k$ for the reason given in Section 3.1 (rounding downward). Now let us assume that $\diamond(x/y) > x/y$. We have $\lfloor \diamond(x/y) \rfloor \geq \lfloor x/y \rfloor$. Then we just need to prove that $\lfloor \diamond(x/y) \rfloor \leq \lfloor x/y \rfloor$, i.e. $\diamond(x/y) < k + 1$.

First note that $\diamond(x/y)$ cannot be a subnormal, since $x/y > 1$. We look for a "good" integer E such that we can write:

- $x = X \cdot 2^E$, where X is an integer;
- $y = Y \cdot 2^E$, where Y is an integer;

and we have $X/Y = x/y$.

If y or $x-y$ has the same exponent as x, then we can take $E = \log_2(\mathrm{ulp}(x))$. Otherwise we take $E = \log_2(\mathrm{ulp}(x)) - 1$. Explanations are given below.

Before analyzing these subcases separately, let us define $d \geq 0$ as the difference between the exponents of x and y. Since x/y is rounded to nearest, we have: $\diamond(x/y) - x/y \leq \mathrm{ulp}(x/y)/2$, as in the closed binade of x/y, the difference between two consecutive floating-point numbers is equal to $\mathrm{ulp}(x/y)$. If two floating-point numbers have the same exponent, their quotient is less than 2; thus by definition of d, we have: $x/y < 2^{d+1}$, then $\mathrm{ulp}(x/y) \leq \mathrm{ulp}(2^d) = 2^{d-n+1}$. Therefore

$$\diamond(x/y) - x/y \leq 2^{d-n}.$$

4.3.1 Subcase $\boldsymbol{E = \log_2(\mathrm{ulp}(x))}$

First, let us prove that X and Y are integers. By definition of ulp, x is an integer multiple of $\mathrm{ulp}(x)$, and X is this integer. If y has the same exponent as x, then $\mathrm{ulp}(y) = \mathrm{ulp}(x)$, and Y is also an integer. If $x-y$ has the same exponent as x, then $\mathrm{ulp}(x-y) = \mathrm{ulp}(x)$, therefore $X-Y$ is an integer, and so is $X-(X-Y)$, i.e. Y.

We recall that we have: $\diamond(x/y) - x/y \leq 2^{d-n}$. Since $X < 2^n$, we have $Y < 2^{n-d}$, and $2^{d-n} < 1/Y$. Therefore $\diamond(x/y) - x/y < 1/Y$.

We have $X/Y = x/y < k+1$, therefore $(k+1)Y - X > 0$, and since it is an integer, $(k+1)Y - X \geq 1$. Therefore $1/Y \leq k+1-x/y$, and as a consequence, $\diamond(x/y) - x/y < k+1-x/y$, i.e. $\diamond(x/y) < k+1$.

4.3.2 Subcase $\boldsymbol{E = \log_2(\mathrm{ulp}(x)) - 1}$

First, let us explain this choice for E. If neither y nor $x-y$ has the same exponent as x, choosing $E = \log_2(\mathrm{ulp}(x))$ may yield a non-integer value for Y. Therefore we need a smaller value for E. Let us show that $\log_2(\mathrm{ulp}(x)) - 1$ is sufficient. First, X is obviously an integer. Let us consider Y. Amongst y and $x-y$, at least one of them has an exponent equal to the exponent of x minus 1, and its ulp is equal to $\mathrm{ulp}(x)/2$. Therefore, amongst Y and $X-Y$, at least one of them is an integer. But this means that both are integers; indeed, since X is an integer, if $X-Y$ is an integer, then $Y = X-(X-Y)$ is also an integer.

Now, as in the subcase $E = \log_2(\mathrm{ulp}(x))$, we can find similar bounds: Since $X < 2^{n+1}$, we have $Y < 2^{n+1-d}$, and $2^{d-n} < 2/Y$. Therefore $\diamond(x/y) - x/y < 2/Y$. We have an upper bound that is twice as large as in the subcase $E = \log_2(\mathrm{ulp}(x))$: it is not possible to end the proof in the same way. And we cannot reduce this bound to $1/Y$, as shown on the following example: $n = 5$, $x = X = 44$, $y = Y = 19$, for which the binary expansion of x/y is $10.010100001\ldots$, and $\diamond(44/19) = 10.011_2 = 19/8$; but $\diamond(44/19) - 44/19 = 9/152 > 8/152 = 1/19$ (our $1/Y$).

We can also consider $r = (k+1)Y - X$, which is a positive integer. If $r \geq 2$, then we can end the proof as in the first subcase: $2/Y \leq k + 1 - x/y$, leading to $\diamond(x/y) < k + 1$.

The remaining case $r = 1$ can occur: $n = 5$, $x = X = 32$, $y = Y = 11$. But we have $x/y < 2^d$ (here, $d = 2$). In such a case, $\mathrm{ulp}(x/y) \leq \mathrm{ulp}(2^{d-1}) = 2^{d-n}$, and $\diamond(x/y) - x/y \leq \mathrm{ulp}(x/y)/2 \leq 2^{d-n-1} < 1/Y \leq k + 1 - x/y$, hence $\diamond(x/y) < k + 1$.

The remaining case is now $r = 1$ and $x/y \geq 2^d$. We have: $2^d \leq x/y < k + 1$, therefore $2^d \leq k$. Then $2^d Y \leq kY = X - Y + 1 < 2^n + 1$, as $2^n \leq X < 2^{n+1}$ and $x - y$ does not have the same exponent as x. Therefore $2^d Y \leq 2^n$. Moreover, by definition of d, we have $2^{n-d} \leq Y < 2^{n-d+1}$, thus $2^n \leq 2^d Y$. As a consequence, Y is a power of two. Since X is even and r is odd, $(k+1)Y$ is odd, and the only possibility is $Y = 1$ and $X = 2^n$. So, we have: $X/Y = 2^n$, which is representable, and $\lfloor \diamond(x/y) \rfloor = \lfloor x/y \rfloor$.

5 The Effect of an Intermediate Extended Precision

The IEEE-754 standard requires that the arithmetic operations shall be correctly rounded to the destination, but does not specify what the destination is. For instance, it may be a floating-point register with a full 64-bit mantissa (as on x86 processors in their default configuration under Linux). But the value could be later stored into memory in double precision (i.e. with a 53-bit mantissa), causing a second rounding.

For the (partial or complete) reproducibility of the results and platform independence, some languages (e.g. Java, ECMAScript, XPath 1.0) forbid extended precision (all results must be correctly rounded to the IEEE-754 double precision). But other languages, like C [2, Section 5.2.4.2.2], allow computations to be performed in a precision larger than the one required by the floating-point type of the result. In such a case, one can have two possible behaviors[3] for Algorithm 1:

- The value of x/y in extended precision is directly used by the floor function. This means that the expression is computed as if the only precision were the extended one. As a consequence, the results of Section 3 can be applied, and we will not consider this case any longer.
- The value of x/y in extended precision is converted to the target precision before the floor function is applied: we have a *double rounding*.

Now, let us analyze the effect of the double rounding in each rounding mode.

5.1 Directed Rounding Modes

It is well known that the double rounding is equivalent to a single rounding in directed rounding modes. This is due to the fact that the floating-point numbers of the target system form a subset of the extended-precision floating-point numbers. So, we always get $\lfloor x/y \rfloor$ in the rounding downward mode; and if x and y are positive integers less than 2^{n-1}, then we also get $\lfloor x/y \rfloor$ in any rounding mode.

5.2 Rounding to Nearest

It is also well known that the double rounding may have unwanted effects in the rounding to nearest mode as the cumulated error of the two roundings may be greater than $\frac{1}{2}$ ulp (which is the maximum error when there is no extended precision). First, let us show that on a simple example. Assume that $n = 5$, $p = 8$, $x = 44$ and $y = 15$, where p denotes the mantissa size in extended precision. In binary, $44/15$ is written $10.111\,011\,101110\ldots$, so that the first rounding (to precision p) yields $\diamond_p(44/15) = 10.111\,100_2$ and the second rounding (to precision n) yields $\diamond_n(\diamond_p(44/15)) = 11.000_2 = 3$ if the rounding-to-even-mantissa rule (i.e. the most common one, like in the IEEE-754 standard, in particular) has been chosen. Therefore, one gets the result 3 instead of 2.

[3]We still assume that the result of the floor function is exactly representable in the target precision, so that even if the floor function produces a value in extended precision, converting it to the target precision does not lead to any rounding. And of course, the input numbers x and y are assumed to be exactly representable in the target precision.

More generally, the counter-examples presented at the end of Section 3 can be reused here, with one more bit for the target precision (as $x - y$ must be exactly representable): $n \geq 3$, $y = 2^{n-1} - 1$ and $x = 3y - 1$. Then $x/y = 3 - 1/y$, where:

$$2^{1-n} < 1/y < 2^{1-n} + 2^{3-2n}$$

(proof: multiply the three terms by y). So, if $p \leq 2n - 2$, then $1/y < 2^{1-n} + 2^{1-p}$, and $\diamond_p(x/y) = 3 - 2^{1-n}$; if moreover, the rounding-to-even-mantissa rule is chosen, then $\diamond_n(\diamond_p(x/y)) = 3$, whereas $\lfloor x/y \rfloor = 2$.

In particular, on an x86 processor using the traditional floating-point unit (no SSE2), the C expression `floor(x/y)` may give 3 instead of 2 for $x = 3 \times 2^{52} - 4$ and $y = 2^{52} - 1$, both of type `double`; the result depends on the compiler optimizations.

We have given counter-examples for $p \leq 2n - 2$, where $n \geq 3$. Let us prove that for any $n \geq 3$ and $p \geq 2n - 1$, the equality $\lfloor \diamond_n(\diamond_p(x/y)) \rfloor = \lfloor x/y \rfloor$ is always satisfied.

In the case $x < y$, we have $x/y \leq 1 - 2^{-n}$, which is exactly representable (see Section 4.1). Therefore $\diamond_p(x/y) \leq \diamond_p(1 - 2^{-n}) = 1 - 2^{-n}$ since $p \geq n$, and $\diamond_n(\diamond_p(x/y)) \leq \diamond_n(1 - 2^{-n}) = 1 - 2^{-n} < 1$. Hence $\lfloor \diamond_n(\diamond_p(x/y)) \rfloor = 0 = \lfloor x/y \rfloor$. In the case $x = y$, we have $\lfloor \diamond_n(\diamond_p(x/y)) \rfloor = 1 = \lfloor x/y \rfloor$. So, if $x \leq y$, the equality is satisfied (even if $p < 2n - 1$, but this is not really useful).

Now, let us assume that $x \geq y$. The error after the double rounding is bounded above by $\varepsilon = \frac{\mathrm{ulp}(x/y)}{2}(1 + 2^{n-p})$. Indeed, the numbers x/y, $\diamond_p(x/y)$ and $\diamond_n(\diamond_p(x/y))$ all belong to a same closed binade $[2^e, 2^{e+1}]$, and when one rounds a value from this binade to nearest in some given precision q, the error is bounded above by $2^{e-q} = \frac{\mathrm{ulp}(x/y)}{2} \times 2^{n-q}$. Let us improve the results of Section 4.3 in both subcases.

- **Subcase $E = \log_2(\mathrm{ulp}(x))$** (Section 4.3.1). We have $Y \leq 2^{n-d} - 1$, then $1/Y > 2^{d-n}(1 + 2^{d-n})$, and $1/Y > \varepsilon$ if $d - n \geq n - p$, i.e. $p \geq 2n - d$. Therefore, if $d \geq 1$, then $p \geq 2n - d$, and $k + 1 - x/y \geq 1/Y > \varepsilon$; this means that $\lfloor \diamond_n(\diamond_p(x/y)) \rfloor < k + 1$, i.e. $\lfloor \diamond_n(\diamond_p(x/y)) \rfloor = k$. If $d = 0$, then $k = 1$ and $X/Y \leq (2^n - 1)/2^{n-1} = 2 - 2^{1-n}$, which is exactly representable; hence $\diamond_n(\diamond_p(x/y)) \leq 2 - 2^{1-n}$, and $\lfloor \diamond_n(\diamond_p(x/y)) \rfloor = 1 = \lfloor x/y \rfloor$.

- **Subcase $E = \log_2(\mathrm{ulp}(x)) - 1$** (Section 4.3.2). As in Section 4, we consider $r = (k+1)Y - X$. If $r \geq 3$, then $k+1-x/y \geq 3/Y > 3 \times 2^{d-n-1} = 2^{d-n}(1+2^{-1})$, which is greater or equal to ε if $n - p \leq -1$; since $p \geq n + 1$, the inequalities are satisfied.

Now, let us assume that $r < 3$, i.e. $r = 1$ or $r = 2$. We recall that we have $Y \leq 2^{n+1-d} - 1$. As a consequence, $1/Y > 2^{d-n-1}(1 + 2^{d-n-1})$. If $r = 2$, then $k + 1 - x/y = 2/Y > \frac{\mathrm{ulp}(x/y)}{2}(1 + 2^{d-n-1})$. If $r = 1$ and $x/y \geq 2^d$, we proved in Section 4.3.2 that in this case, y was a power of two, so that x/y was exactly representable and $\lfloor \diamond_n(\diamond_p(x/y)) \rfloor = \lfloor x/y \rfloor$; this case does not need further discussion. So, if $r = 1$, we assume that $x/y < 2^d$, from which we have $\mathrm{ulp}(x/y) \leq 2^{d-n}$; therefore $k + 1 - x/y = 1/Y > \frac{\mathrm{ulp}(x/y)}{2}(1 + 2^{d-n-1})$.

So, in our remaining cases, $k + 1 - x/y > \frac{\mathrm{ulp}(x/y)}{2}(1 + 2^{d-n-1})$. If $d \geq 2$, then $d - n - 1 \geq 1 - n \geq n - p$, as $p \geq 2n - 1$; therefore $k + 1 - x/y > \varepsilon$. Otherwise $d = 1$ (we recall that y does not have the same exponent as x).

- If $x/y < 2$, then $k = 1$, and as X is even, r is also even, i.e. $r = 2$. As a consequence, $k + 1 - x/y = 2/Y > 2^{1-n} \geq \mathrm{ulp}(x/y) \geq \varepsilon$.
- If $x/y \geq 2$, then $k \geq 2$ and $r = 2$. Therefore $X - Y = kY - 2 \geq 2Y - 2$. In this subcase, $X - Y$ does not have the same exponent as X; since $d = 1$, $2Y$ has the same exponent as X, and $2Y - 2$ does not have the same exponent as $2Y$. As a consequence, Y is a power of two, and x/y is exactly representable; hence $\lfloor \diamond_n(\diamond_p(x/y)) \rfloor = \lfloor x/y \rfloor$.

So, we have the following theorem. Note that the considered exponent range is the one of the target system (in precision n).

Theorem 2 *Let n and p be two positive integers such that $n \geq 3$ and $p \geq 2n - 1$. Consider a floating-point system in base 2 and internal precision p, with or without subnormals, such that $1/2$ is a normal representable number. If u is a real number, let $\diamond_n(u)$ and $\diamond_p(u)$ denote the floating-point numbers, with n-bit and p-bit mantissas respectively, that are the nearest to u (in halfway cases, any of the enclosing floating-point numbers can be chosen). Let x and y be n-bit floating-point numbers such that $x \geq 0$, $y > 0$ and $x - y$ is exactly representable with an n-bit mantissa. Assume that x/y does not overflow. Then $\lfloor \diamond_n(\diamond_p(x/y)) \rfloor = \lfloor \diamond_n(x/y) \rfloor = \lfloor x/y \rfloor$, and this value is exactly representable.*

6 Conclusion

We studied conditions under which the Euclidean division between two nonnegative numbers of a binary floating-point system can be implemented with a floating-point division followed by a floor (or trunc) function. We had to assume a few properties on the floating-point system:

- If the arithmetic has an intermediate extended precision p when rounding to nearest is used, then p must be greater or equal to $2n - 1$, when n is the precision of the target system. Basically, this means that the processor rounds the results to precision p, and a second rounding is performed whenever a result is stored to memory. Unfortunately, the condition $p \geq 2n - 1$ is not satisfied for x86 processors configured to round in extended precision (64-bit mantissas). But one should note that some languages (e.g. Java, ECMAScript, XPath 1.0) forbid extended precision[4], and when this is allowed (e.g., in ISO C), one can generally change the rounding mode, and possibly the rounding precision (however not in a portable way, unfortunately). One can also notice that the double precision (53 bits) may be used as an extended precision for a target in single precision (24 bits), and the quadruple precision (113 bits) may be used as an extended precision for a target in double precision.
- The division must be correctly rounded. This is the case in any reasonable floating-point implementation, in particular with IEEE-754 conforming implementations.
- The exponent range must be large enough. Again, this is the case in any reasonable floating-point implementation, in particular with IEEE-754 conforming implementations.

The results are summarized here, depending on the rounding mode:

- **Rounding to nearest:** It is required by the IEEE-754 standard, where it must be the default rounding mode; moreover it is even the *only* rounding mode in some languages. Here an implementation using a floating-point division followed by a floor function generally leads to the correct result in practical cases, where floating-point is used to represent integers. There is special benefit for languages without an integer division (e.g. ECMAScript, XPath 1.0), but this is still interesting in any other language. Unfortunately, the result is not necessarily correct in the very general case.

[4] However many implementations do not conform to the standards and work in extended precision internally.

- **Rounding downward:** The result is always correct. This is particularly useful in the very general case.
- **Rounding upward:** The only interest is to avoid a change of the rounding mode in some contexts, when rounding modes are dynamical. Unfortunately, simple cases may fail to give the correct result, so that this rounding mode should be avoided if possible.

When the numbers can be negative, the Euclidean division can have two possible definitions, leading to different results: the floor function may be used so that the remainder is nonnegative, or the trunc function may be used to get the smaller integer in absolute value. In the latter case and rounding toward 0 or to nearest, the above results still hold for symmetry reasons.

7 Acknowledgments

I would like to thank Paul Zimmermann for his suggestions.

References

[1] ECMA. *ECMA-262: ECMAScript Language Specification.* ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[2] International Organization for Standardization. *ISO/IEC 899:1999: Programming Languages — C.* International Organization for Standardization, Geneva, Switzerland, December 1999.

[3] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991. An edited reprint is available at `http://cch.loria.fr/documentation/IEEE754/ACM/goldbergSUN.ps` from Sun's Numerical Computation Guide; it contains an addendum *Differences Among IEEE 754 Implementations*, also available at `http://cch.loria.fr/documentation/IEEE754/ACM/addendum.html`.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, second edition, 1997.
http://java.sun.com/docs/books/jls/

[5] IEEE standard for binary floating-point arithmetic. Technical Report ANSI-IEEE Standard 754-1985, New York, 1985. Approved March 21, 1985: IEEE Standards Board, approved July 26, 1985: American National Standards Institute, 18 pages.

[6] V. Lefèvre. An algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$. In *Developments in Reliable Computing*, pages 203–212. Kluwer, Dordrecht, Netherlands, 1999.

[7] V. Lefèvre. New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, June 2005. To appear.
http://arith17.polito.it/final/paper-147.pdf

[8] XML path language (XPath) version 1.0, W3C recommendation, November 1999.
http://www.w3.org/TR/xpath

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399