Received: 5 March 2022 | Revised: 5 October 2022 | Accepted: 17 October 2022

DOI: 10.1002/spe.3172

RESEARCH ARTICLE

WILEY

Euclidean affine functions and their application to calendar algorithms

Cassio Neri[1] | **Lorenz Schneider**[2]

[1]Independent Researcher, London, UK

[2]EMLYON Business School, Lyon, France

Correspondence
Lorenz Schneider, EMLYON Business School, Lyon, France.
Email: schneider@em-lyon.com

Abstract

In everyday life, dates are specified in terms of year, month and day, but this is not how digital devices represent them. Such devices continuously count elapsed days since a certain reference date, usually 1 January 1970. Accordingly, the date exactly one year after this reference is 1 January 1971 and digital devices represent it as 365. Conversions between machine and human formats are, arguably, amongst the most common operations performed by digital devices and constitute the subject of this article. We introduce Euclidean affine functions (EAFs) and study their properties. EAFs are of the form $f(n) = (a \cdot n + b)/d$, where n, a, b, and d are integers and where $/$ denotes the quotient of Euclidean division. We derive algebraic relations and numerical approximations that are important for the efficient evaluation of these expressions in modern CPUs. Since division is a particular case of an EAF (when $a = 1$ and $b = 0$), the optimisations proposed in this article can also be applied to division. The main application presented in this article is the derivation of conversion algorithms for the Gregorian calendar. We will show that they can be implemented substantially more efficiently than is currently the case in widely used C, C++, C#, and Java open source libraries. Gains in speed of a factor of two or more are common. These algorithms have been implemented in GCC, the Linux Kernel and .NET.

KEYWORDS

Euclidean affine functions, integer division, calendar algorithms

1 | INTRODUCTION

The most widely used civil calendar today, adopted by almost every country in the world, is the Gregorian calendar, named after Pope Gregory XIII who introduced it in 1582. At that time, the standard calendar in Europe was the Julian calendar, brought in by Julius Caesar, the supreme ruler of Rome, in 45 BC.* These two calendars are very similar and define exactly the same months, from January to December. Months have the same (varying) number of days in both calendars. Years

*The fascinating history of calendars is beyond the scope of this article. Interested readers are directed to Richards[1] and Duncan.[2]

Abbreviations: EAF, Euclidean Affine Function; CPU, Central Processing Unit; GCC, The Gnu Compiler Collection.
[Correction added on 21 December 2022, after first online publication: the expanded form of CPU was corrected in this version.]

usually have 365 days, but some – called **leap years** as opposed to **common years** – have an extra day, making them 366 days long. A key difference between the Julian and Gregorian calendars, however, is the rule used to determine leap years, as stated below.

Definition 1 (Julian rule.). A **Julian leap year** is a multiple of 4.

Definition 2 (Gregorian rule.). A **Gregorian leap year** is a multiple of 4, except if it is divisible by 100 but not by 400.

The next year that these rules will disagree on is 2100, a leap year in the Julian calendar but not in the Gregorian calendar. Since 2000 was a leap year in both calendars, the most recent year they disagreed on was 1900. Generally, these rules diverge three times in any span of 400 years, so that in such periods there are 100 Julian leap years but only 97 Gregorian leap years.

The reason for introducing leap years is, roughly speaking, to make the average calendar year closer to an actual astronomical value. A year[†] is approximately 365.2424 days long, that is, a bit longer than the 365 days ascribed to common years. Increasing the calendar year every now and then brings the average duration closer to 365.2424. Accordingly, in each 4-year period of the Julian calendar, there are 3 common years and 1 leap year. Hence, the average duration of the Julian year is:

$$\frac{3 \cdot 365 + 1 \cdot 366}{4} = \frac{1461}{4} = 365.2500.$$

This entails an error of $365.2500 - 365.2424 = 0.0076$ days per year, which might not seem too large, but which amounts to 1 day every $0.0076^{-1} \approx 132$ years. Unsurprisingly, 16 centuries after its introduction, the misalignment between the Julian calendar and the March equinox was noticeable enough to motivate the Gregorian reform.

Similarly, in each 400-year period of the Gregorian calendar, there are 303 common years and 97 leap years. Hence, the average duration of the Gregorian year is:

$$\frac{303 \cdot 365 + 97 \cdot 366}{400} = \frac{146097}{400} = 365.2425.$$

The error is $365.2425 - 365.2424 = 0.0001$ days per year, or 1 day every $0.0001^{-1} = 10{,}000$ years.[‡]

We are primarily interested in the Gregorian calendar, given its widespread use today. We nonetheless first derive algorithms for the Julian calendar as the process for that calendar is simpler, providing easier examples of our results. For the same reason, we in fact consider the Egyptian calendar[§] first of all, before either the Julian or Gregorian calendars. The Egyptian calendar has no leap year, so every year is 365 days long. Conversions for this calendar are covered in Section 2, but it is no surprise that they involve division by 365. We now review previous results on division by constants before considering the generalisations required to tackle the conversions for the Julian and Gregorian calendars.

Divisions by constants appear in some of the most common tasks performed by software systems, including printing decimal numbers (division by powers of 10) and working with times (division by 24, 60, and 3600). Since division is the slowest of the four basic arithmetical operations, various authors[4-8] have proposed strength reduction optimisations (i.e., the replacement of instructions with alternatives that are mathematically equivalent but faster) for integer divisions when divisors are constants known by the compiler. The algorithms proposed by Granlund and Montgomery[6] have been implemented by major compilers, for example.

Remainder calculation is a closely related problem, also arising in the tasks mentioned above. Nevertheless, the cited works do not consider optimisations for this operation. Even when only the remainder is necessary but not the quotient, compilers are content to apply strength reduction to obtain the quotient $q = n/d$ first and then evaluate the expression

[†] This is the number of days between two consecutive March equinoxes. This is just one of many related, but different, astronomical definitions that in layman's terms is referred to as a *year*. The astronomy underlying calendars is yet another fascinating topic that is also beyond the scope of our paper. For details, see Steel.[3]

[‡] Although this clearly indicates the superiority of the Gregorian calendar over the Julian calendar in terms of aligning with the March equinox, this number must be taken with a pinch of salt. Firstly, for the sake of simplicity we are not showing enough decimal digits and so the accuracy suffers. Secondly, the duration of the year (in SI seconds) is not constant (and neither is the duration of a day). Over such a large time-span, it changes noticeably. It is therefore difficult to obtain accurate results using currently known values.

[§] Actually, the *Egyptian civil calendar*, to distinguish it from other calendars the ancient Egyptians used. For brevity, we shall simply refer to it as the *Egyptian calendar*. The exact date this calendar was introduced is not known, but it is speculated to be some time between 2937 and 2821 BC.[1]

$r = n - d \cdot q$, which gives the remainder $r = n\%d$. For this reason, Lemire et al.[9] and Warren[10] considered the problem of directly obtaining the remainder without first calculating the quotient.

Due to the variable lengths of years and months, conversions from elapsed days (since a fixed reference date) to dates in the Julian and Gregorian calendars cannot be tackled with a simple sequence of quotient and remainder calculations, at least not in the same form as seen for the Egyptian calendar. The method is nonetheless fairly similar. Indeed, as we shall see in this article, this pattern is recovered in a more general setting that uses functions of the form $f(n) = (a \cdot n + b)/d$ – called Euclidean affine functions (EAFs) – of which division is a particular case (when $a = 1$ and $b = 0$).

The previous paragraph touches on a critical point that distinguishes implementations of calendar algorithms: how they deal with variable month and year lengths. Some implementations[11,12] resort to look-up tables, which can be costly when the L1-level cache is cold and can increase cache thrashing since the implementations compete for cache memory against other running software. The implementations conduct linear searches on the look-up tables, entailing branching that can cause stalls in the processor's execution pipeline. Other implementations[1,13-19] tackle the issue entirely through EAFs. Nevertheless, they do not go far enough and do not use the mathematical properties derived in this article. To the best of our knowledge, the closest work to make use of some of these properties is Baum.[13] He provides some explanations, but appears to resort to trial and error when it comes to pre-calculating certain *magic numbers* used in his algorithm. We go further by providing a systematic and general framework for such calculations.

This article expands previous works in two ways. Firstly, it considers the more general setting of EAFs. Secondly, it suggests optimisations that are even more effective in applications where both the quotient and the remainder need to be evaluated.

We derive EAF-related equalities that provide alternative ways of evaluating expressions commonly used in applications. (For instance, $n - d \cdot (n/d)$, which is used to obtain the remainder as explained above.) These equalities underpin optimisations, other than strength reduction, that take into account aspects of modern CPUs. Specifically, they foster instruction-level parallelism implemented by superscalar processors and they profit from the backward compatibility features that drove the design of the x86_64 instruction set.

Our Gregorian calendar algorithms are substantially faster than those in widely used open source implementations, as shown by benchmarks against counterparts in glibc,[11,20] Boost,[17] libc++,[18] .NET,[12][¶] and OpenJDK[21] (Android contains the same implementations).[22] Our algorithms are also faster than others found in the academic literature.[1,13-16,19]

Our paper and its main contributions are organised as follows.

Section 2 covers conversions for the Egyptian calendar, pinpointing the mathematical properties of division that need to be generalised for EAFs in order to enable usage with more complex calendars. This section also motivates a *twist of the calendar*, the topic of Section 4.

Section 3 introduces EAFs and states some of their properties. Their proofs are set out later in Section 13. Although EAFs appear in competitor algorithms, we were unable to find any systematic coverage of them. Hatcher[15] and Richards[1] appear to be aware of some of the results of Theorem 1, but they do not point to any proof.

Sections 5 and 6 make use the results of Sections 3 and 4, and derive the conversion algorithms for the Julian and Gregorian calendars, respectively.

Section 7 concerns efficient evaluations of EAFs. Theorems 2 and 3 generalise prior-known results on division to EAFs. The proofs are shown in Section 14, which revisits previous results and brings geometric insights to the problem. Theorem 5 concerns the efficient evaluation of residuals: they are to EAFs what remainders are to division. The optimisation proposed by Theorem 5 is fundamentally different from other optimisations, both in the present article and in prior works, since it does not involve strength reduction. Indeed, this theorem shows how to break a data dependency present in the instructions currently emitted by compilers, enabling instruction-level parallelism, as we shall see in Section 8.2.

Section 8 applies the results of Section 7 to obtain optimised, but theoretical, versions of the algorithms for the Gregorian calendar. Sections 9 and 10 cover practical aspects that we take into consideration in Section 11 to derive practical optimised implementations of the algorithms. Finally, Section 12 presents the performance analysis.

We conclude this introduction by specifying the notation used throughout the article and by recalling some well-known results.

[¶] During the revision of this article, we were informed about a .NET pull request that implements one of our algorithms. It is expected to be part of release 7.

Forward slash / and percent % respectively denote the **quotient** and **remainder** of Euclidean division. More precisely, given n, $d \in \mathbb{Z}$, with $d \neq 0$, there exist unique q and $r \in \mathbb{Z}$ such that $0 \leq r < |d|$ and $n = d \cdot q + r$. Then $n/d = q$ and $n\%d = r$.

We give multiplication, quotient and remainder operators equal precedence. Hence, $a \cdot b\%c = (a \cdot b)\%c$, $a\%b \cdot c = (a\%b) \cdot c$, $a/b\%c = (a/b)\%c$ and $a\%b/c = (a\%b)/c$. Moreover, $a + b\%c = a + (b\%c)$, and $a - b\%c = a - (b\%c)$.

The set of non-negative integer numbers is denoted by $\mathbb{Z}^+ = \{x \in \mathbb{Z}\ ;\ x \geq 0\}$. We will occasionally work on non-integer numbers, but they will all be rationals, that is, elements of $\mathbb{Q}$. It is important to distinguish the Euclidean quotient n/d (an integer number) from rational division $n \cdot d^{-1} = \frac{n}{d}$ (a rational number). Obviously, if $n \cdot d^{-1}$ is an integer, then $n/d = n \cdot d^{-1}$.

For any $x \in \mathbb{Q}$, $\lfloor x \rfloor$ and $\lceil x \rceil$ respectively denote the **floor of** x – the largest integer not larger than x – and the **ceil of** x – the smallest integer not smaller than x.

A date is a triplet $X = (Y, M, D)$ where Y, M and D are the integer values of **year**, **month** and **day**. For the Gregorian calendar, it is convenient to break Y down into two components: the **century** C and the **year of the century** Z, related by:[#]

$$Y = 100 \cdot C + Z, \qquad C = Y/100 \qquad \text{and} \qquad Z = Y\%100.$$

Comparisons between dates have chronological meaning. For instance, $X_1 < X_2$ means that X_1 is before X_2 and $X_1 \geq X_2$ means that X_1 is after or on X_2. For $X_1 \leq X_2$, the interval $[X_1, X_2[$ is referred to as **the days from** X_1 **to** X_2 or simply **the days up to** X_2 when X_1 is implied from the context. The number of elements of this interval is denoted by $\#[X_1, X_2[$. This definition is extended to the case $X_1 > X_2$ by $\#[X_1, X_2[\ := -\#[X_2, X_1[< 0$. (A negative number of days represents backwards counting.) With a slight abuse of language, in this case we still refer to **days from** X_1 **to** X_2 or **days up to** X_2 to refer to the interval $[X_2, X_1[$.

The **epoch** is a fixed reference date X_0 set by the context and from which days are counted. For any date X, the number of days from X_0 to X is called its rata die.[‖]

Let X_0 be a fixed epoch on a given calendar. The conversion problems of interest are formalised as follows. Given a date X find its *rata die* N. Conversely, given N find the date X whose *rata die* is N.

2 ALGORITHMS FOR THE EGYPTIAN CALENDAR

In this ancient calendar, every year has 365 days split into 13 months numbered from 0 to 12.[**] The first 12 months have 30 days each, totalling 360, and the last month contains the remaining 5 days. In each month, days are numbered from 0. Conversions on this calendar are very similar to time conversions where multiplication by 60 always converts hours to minutes and, conversely, division by 60 always converts minutes to hours.

Deriving algorithms for this calendar is therefore as straightforward as working with times. However, we follow a path that introduces concepts and notations requiring generalisations for usage with more complex calendars. The Egyptian calendar provides concrete examples that we can retain in this first encounter with the concepts of interest.

In this section, the epoch is set to $X_0 = (0, 0, 0)$. Our search for a date $X = (Y, M, D)$ whose *rata die* is a given N starts with its year Y. We introduce the following functions:

$y(N) :=$ **the year** Y. For the Egyptian calendar, $y(N) = N/365$.

$y^\circ(N) :=$ **the day of the year**, that is, the number of days from the first day of year Y to X. For the Egyptian calendar, $y^\circ(N) = N\%365$ – the circle ∘ is a mnemonic referring to the circles in %.

[#] An *off-by-one* cautionary note: our definition of century is 0-based, whereas in common usage it is 1-based (forward for AD years and backward for BC years). For instance, the year 2022 belongs to the 21st century but the definition above gives the century $C = 2022/100 = 20$ and the year of the century $2022\%100 = 22$ so that $2022 = 100 \cdot 20 + 22 = 100 \cdot C + Z$.

[‖] The term *rata die* is usually[13,19] applied to the particular case where the epoch is, roughly speaking, 31 December 0000, but we shall use it regardless of the epoch.

[**] Only the first 12 of these 13 periods were considered real months and they had names. The last 5-day period was outside any month and the days were referred to as *epagomenal* days.[1] However, this is not relevant to our calculations: we will refer to all periods as *months* and only consider their numerical values.

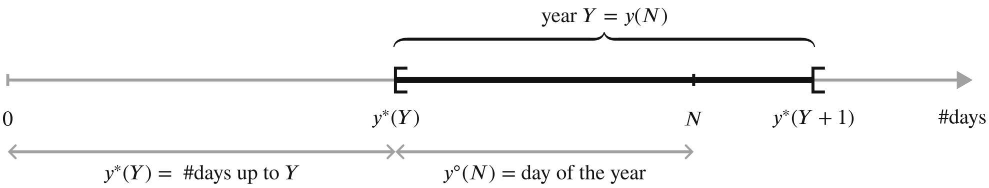


FIGURE 1 Illustration of $Y = y(N)$, $y^\circ(N)$ and $y^*(Y)$

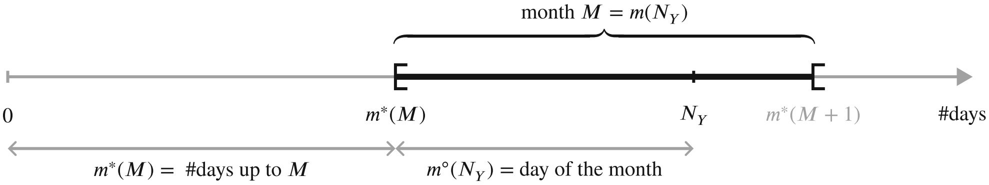


FIGURE 2 Illustration of $M = m(N_Y)$, $m^\circ(N_Y)$ and $m^*(M)$

$y^*(Y)$:= **the number of days up to the first day of year** Y, that is, the number of days in all years in $[0, Y[$. For the Egyptian calendar, $y^*(Y) = 365 \cdot Y$ – the symbol $*$ refers to the product operation.

Figure 1 illustrates these quantities. The axis represents the number of days since the epoch. Hence, 0 symbolises the epoch and N is pictured on this axis. A certain interval of values falls on the same year Y and this is depicted as a heavier line. It starts at $y^*(Y)$ and ends at $y^*(Y+1)$ – the beginning of the following year. The difference $N - y^*(Y)$ is the day of the year $y^\circ(N)$.

By construction, the following holds:

$$y(N) = Y \iff y^*(Y) \leq N < y^*(Y+1), \tag{1}$$

$$y^\circ(N) = N - y^*(y(N)). \tag{2}$$

For the Egyptian calendar, the above reads that $N/365 = Y$ if, and only if, $365 \cdot Y \leq N < 365 \cdot (Y+1)$, and that $N\%365 = N - 365 \cdot (N/365)$. This should come as no surprise. However, let's pretend that we don't know about this result or about the formulas of y, y^* and y°. We only know about Equations (1) and (2), which are generic and apply to all calendars.

Usually, y^* is easier to find than the other functions and after we find it, Eqs. (1) and (2) help to deduce y and y°. For the Egyptian calendar, recall now that $y^*(Y) = 365 \cdot Y$. We easily recognise that $365 \cdot Y \leq N < 365 \cdot (Y+1)$ if, and only if, $Y = N/365$. Hence, Equation (1) gives $y(N) = N/365$. Now Equation (2) yields $y^\circ(N) = N - 365 \cdot (N/365)$ and thus $y^\circ(N) = N\%365$. At this stage, we conclude that if N is the *rata die* of $X = (Y, M, D)$ and N_Y is its day of the year, then

$$Y = y(N) = N/365 \quad \text{and} \quad N_Y = y^\circ(N) = N\%365.$$

To finish, we need to find M and D given the day of the year N_Y. This is done in a very similar way to the previous step and we only provide a quick overview here, rather than providing the full details. We introduce the functions m, m° and m^* as depicted in Figure 2. An important difference with respect to Figure 1 is that 0 now represents the first day of year Y rather than the epoch. In particular, $m^*(M)$ is defined as the number of days in all months in $[0, M[$.

The counterparts to Equations (1) and (2) are:

$$m(N_Y) = M \iff m^*(M) \leq N_Y < m^*(M+1),$$
$$m^\circ(N_Y) = N_Y - m^*(m(N_Y)).$$

From these and m^* we deduce m and m°. For the Egyptian calendar, we have $m^*(M) = 30 \cdot M$ and it follows that:

$$M = m(N_Y) = N_Y/30 \quad \text{and} \quad D = m^\circ(N_Y) = N_Y\%30.$$

It is worth noting that had we numbered the days in a month from 1, the expression $N_Y\%30$ would be off by 1.

Regarding the opposite conversion, from the definitions of y^* and m^*, we find that the *rata die* of X is:

$$N = y^*(Y) + m^*(M) + D.$$

For the Egyptian calendar, this gives $N = 365 \cdot Y + 30 \cdot M + D$.

There is an important point worth mentioning about m^*. By definition, $m^*(M)$ is the number of days up to, but excluding, month M. In particular, $M(12)$ ignores days *in* month 12. These days would be accounted for by $M(13)$, except that 13 is not a valid month and $M(13)$ is thus not defined. (This is why $m^*(M+1)$ is greyed-out in Figure 2: it has no meaning for $M = 12$.)

Astute readers might ask, Since m is deduced from m^*, which is oblivious to month 12, why does $N_Y/30$ yield the right month, even when N_Y represents dates in month 12? What if month 12 had a different number of days instead of 5, would $N_Y/30$ still be correct?

This mystery is solved by the realisation that the expression $N_Y/30$ is not deduced from m^* but from the expression $30 \cdot M$, which is well defined for $M = 13$. It so happens that $30 \cdot M$ and $m^*(M)$ agree up to $M = 12$ and, similarly, that $N_Y/30$ and $m(N_Y)$ agree up to $N_Y = 364$.[††] Month 12 could be longer than its ascribed 5 days provided that N_Y stays below $30 \cdot 13 = 390$ (i.e., provided that the year is not longer than 390 days), otherwise $N_Y/30$ reaches 13 but $m(N_Y)$ cannot do so.

TAKEAWAY POINTS

- We have defined two triplets of functions (y, y^*, y°) and (m, m^*, m°) related to year and month, respectively. Generally, each triplet is denoted by (f, f^*, f°). Conceptually, f acts as a "division" that splits a certain number of days into periods (years or months). Reciprocally, f^* is like a "multiplication" that counts the number of days in a given number of periods. Finally, f° behaves as a "remainder".
- $y^*(Y)$ is the number of days up to year Y and $m^*(M)$ is the number of days up to month M. Generally, $f^*(q)$ is the number of days up to period q. Usually, f^* is simple to deduce from the calendar structure.
- From f^* we have deduced f and f° using:

$$\begin{aligned} f(n) = q \ &\Leftrightarrow\ f^*(q) \le n < f^*(q+1), \\ f^\circ(n) &= n - f^*(f(n)). \end{aligned} \tag{3}$$

- The date $X = (Y, M, D)$ whose *rata die* is a given N is found by:

$$\begin{aligned} Y &= y(N), \quad & N_Y &= y^\circ(N), \\ M &= m(N_Y), \quad & D &= m^\circ(N_Y). \end{aligned}$$

- Reciprocally, the *rata die* of $X = (Y, M, D)$ is given by:

$$N = y^*(Y) + m^*(M) + D. \tag{4}$$

- In each month, days are numbered from 0. (Otherwise, $m^\circ(N_Y)$ would not give the correct day and would be off by 1.)
- m^* does not encode any information about the number of days in the last month.

[††] Recall that N_Y is 0-based and, since the year is 365 days long, $N_Y \in [0,364]$.

3 | EUCLIDEAN AFFINE FUNCTIONS

The simplicity of the Egyptian calendar allowed us to easily find $y(N) = N/365$ once we had $y^*(Y) = 365 \cdot Y$. Unfortunately, the existence of leap years in the Julian and Gregorian calendars forces y^* to take another form that is not just a multiplication, as the following example shows.

Example 1 (y^* for the Julian calendar). Let $y^*(Y)$ be the number of days in all years in $[0, Y[$. Therefore, $y^*(Y)$ must exceed $365 \cdot Y$ by the number of leap years in $[0, Y[$ or, in other words, the number of multiples of 4 in $[0, Y[$. It is relatively easy to deduce (and left to the reader) that the number of multiples of 4 in $[0, Y[$ is $(Y + 3)/4$. Therefore,

$$y^*(Y) = 365 \cdot Y + (Y + 3)/4 = (1461 \cdot Y + 3)/4. \tag{5}$$

The form taken by y^* motivates the next definition.

Definition 3. A function $f : \mathbb{Z} \to \mathbb{Z}$ is a **Euclidean affine function**,[‡‡] **EAF** for short, if it has the form $f(n) = (a \cdot n + b)/d$ for all $n \in \mathbb{Z}$ and fixed $a, b, d \in \mathbb{Z}$ with $d \neq 0$.

When (f, f^*, f°) is a triplet of calendrical functions as in Section 2, these three functions must obey Equation (3). If $f^*(q) = a^* \cdot q$ is just a product by a^*, as for the Egyptian calendar with $a^* = 365$, then it is easy to deduce that $f(n) = n/d$ is the division by $d = a^*$ and that $f^\circ(n) = n\%d$ is the remainder of the same division. As we shall see in Examples 2 and 4 (below), the following theorem covers the more general case where f^* is an EAF.

Theorem 1 (EAF Theorem). *Let $f^*(q) = (a^* \cdot q + b^*)/d^*$ with $a^* \geq d^* > 0$. Set $a = d^*$, $b = d^* - b^* - 1$, $d = a^*$ and*

$$f(n) = (a \cdot n + b)/d,$$
$$f^\circ(n) = (a \cdot n + b)\%d/a.$$

Then, for $n \in \mathbb{Z}$ and $q \in \mathbb{Z}$ we have:

$$f(n) = q \iff f^*(q) \leq n < f^*(q + 1), \tag{6}$$

$$f^\circ(n) = n - f^*(f(n)). \tag{7}$$

Proof. See Section 13. ■

The condition $a^* \geq d^*$ above means that in f^*'s expression, the multiplier is at least as large as the divisor and, conceptually, f^* acts as a multiplication. In contrast, f swaps the roles of a^* and d^*, and behaves like a division.

The next examples interpret the EAF theorem in the context of calendrical calculations, generalising the arguments seen in Section 2.

Example 2 (Year EAF). Let $f^*(q)$ be the number of days in all years from some reference year up to year q. (Similar to y^* of Section 2.) Let $X = (Y, M, D)$ be the date that is n days after the start of this counting. Suppose that f^* is an EAF and let f and f° be as in Theorem 1. We emphasise that f^* has a calendrical meaning and turns out to be an EAF. In contrast, f and f° are simply the functions algebraically defined in the theorem and, up to this point, we ignore any calendrical interpretation they might have.

Looking at the right side of Equation (6), let q be such that $f^*(q) \leq n < f^*(q + 1)$. This means that X is on or after the first day of year q and before the first day of year $q + 1$, or in other words, X is *in* year q, that is, $q = Y$. Equation (6) says that this is equivalent to $f(n) = q = Y$. We have just deduced the calendrical meaning of $f(n)$: it is the year in which a date n days after the start of the counting falls. Moreover, $n - f^*(f(n)) = n - f^*(q)$ is the number of days from the first day of year q up to X or, in other words, it is the day of the year. Therefore, Equation (7) says that this number matches $f^\circ(n)$, that is, $f^\circ(n)$ is the day of the year of the date that is n days after the start of the counting.

[‡‡] This terminology is ours. The analogues of EAFs in higher dimensions appear in Discrete Geometry and are called quasi-affine transformations. That area focuses on periodicity, tiling and other geometric aspects, whereas we are interested in efficient calculations. We therefore use the term EAF to distinguish between these two approaches.

The following is a more concrete version of the above for the Julian calendar.

Example 3 (Year EAF for the Julian calendar). As seen in Example 1, for the Julian calendar, the number of days in years in $[0, Y[$ is $y^*(Y) = (1461 \cdot Y + 3)/4$. Therefore, Theorem 1 for $f^* = y^*$ with constants $a^* = 1461$, $b^* = 3$ and $d^* = 4$ yields $a = 4$, $b = 0$, $d = 1461$ and the functions $y(N) = 4 \cdot N/1461$ and $y^\circ(N) = 4 \cdot N\%1461/4$ respectively give the year and the day of the year of the date N days after the first day of year 0.

What Example 2 did for years, the next one does for months.

Example 4 (Month EAF). Let $f^*(q)$ be the number of days in all months from the first day of the year up to month q. (Similar to m^* of Section 2.) Let $X = (Y, M, D)$ be the date whose day of the year is $N_Y \in [0,365[$. Suppose that f^* is an EAF and let f and f° be as in Theorem 1. The same reasoning used in Example 2 reveals that if $f^*(q) \le N_Y < f^*(q+1)$, then X falls on month q, that is, $q = M$. From Equation (6), this is equivalent to $f(N_Y) = q = M$ and $f(N_Y)$ is thus the month of the date whose day of the year is N_Y. Now, Equation (7) gives $f^\circ(N_Y) = N_Y - f^*(f(N_Y)) = N_Y - f^*(M)$, which is the number of days from the beginning of the month to X, that is, $f^\circ(N_Y) = D$.

To finish this section, we discuss other points motivated by Theorem 1. Its statement presents the equations to obtain a, b and d – the coefficients of f – from a^*, b^* and d^* – those of f^*. Reciprocally, we can use the same equations to solve for a^*, b^* and d^* to obtain $a^* = d$, $b^* = a - b - 1$ and $d^* = a$. Therefore, the coefficients of f can be used to find those of f^*. Furthermore, given $q \in \mathbb{Z}$, Equation (6) states that the set of solutions of $f(n) = q$ is an interval whose smallest element is $f^*(q)$ or, in other words, $n = f^*(q)$ is the minimal solution of $f(n) = q$. This discussion motivates the following two definitions.

Definition 4. The **minimal right inverse** of $f(n) = (a \cdot n + b)/d$, with $d \ge a > 0$, is defined by $f^*(q) = (a^* \cdot q + b^*)/d^*$, where $a^* = d$, $b^* = a - b - 1$ and $d^* = a$.

Definition 5. The **residual function** of $f(n) = (a \cdot n + b)/d$ is defined by $f^\circ(n) = (a \cdot n + b)\%d/a$.

4 | THE COMPUTATIONAL CALENDAR

February – the second month of the Julian and Gregorian calendars – is the ugly duckling of the months. It is shorter than the others and, worse, its number of days depends on whether the year is a leap year or a common year and thus $m^*(M)$ – the number of days in all months up to M – cannot be the same for leap and common years. For this reason, some implementations[12,20] resort to two look-up tables, one for leap years and another for common years, to store the values of $m^*(M)$ for $M \in \{1, \dots, 12\}$. As we shall see in this section, by a twist of the calendar, we get a new calendar where m^* can be stored by a single table. Even better, we do not need a table at all.

As discussed in Section 2, m^* does not encode any information on the last month. In addition, provided it is not too long, the last month can have different numbers of days. This suggests that things would be easier if February were the last month of the year.[§§] Fortunately, as far as algorithms are concerned, we do not need to have the powers of a Roman dictator or a Pope to "reform" the calendar and move February to the end of the year. Indeed, this is what German mathematician Christian Zeller[23] did in 1882 for his algorithm (known as Zeller's congruence), which finds the day of the week for any given date.[¶¶] He introduced a mathematically convenient calendar that is easily mapped to the Julian and Gregorian calendars. Borrowing Hatcher's[15,16] terminology, we call it the **computational calendar**.

In the computational calendar, December of year Y is followed by January and February of the same year (in contrast to the Gregorian/Julian calendars in which they fall in year $Y + 1$). Accordingly, January and February are months 13 and 14 of year Y. The following March is the first month of year $Y + 1$.

Figure 3 illustrates the relationships between the Gregorian/Julian and computational calendars for years 0 and 1. There is no difference from March ($M = M_{G/J} = 3$) to December ($M = M_{G/J} = 12$). For February, $M_{G/J} = 2$ of year 1 (generally, $Y + 1$) of the Gregorian/Julian calendar corresponds to $M = 14$ of year 0 (generally, Y) of the computational calendar. Therefore, the computational calendar's leap year rule is off-by-one with respect to the Gregorian/Julian calendars. In other words, Y is a leap year in the computational calendar if, and only if, $Y + 1$ is a leap year in the corresponding

[§§] According to some accounts,[1] this was the case until the *Decemviri* – the ten men who first penned the Roman code of laws – moved February to its current position in 450 BC.

[¶¶] In essence, his algorithm calculates the *rata die* modulus 7.

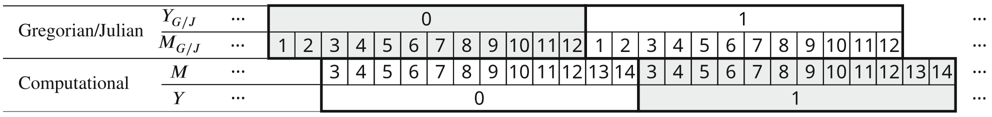

Gregorian/Julian
$Y_{G/J}$ ··· 0 1 ···
$M_{G/J}$ ··· 1 2 3 4 5 6 7 8 9 10 11 12 1 2 3 4 5 6 7 8 9 10 11 12 ···
Computational
M ··· 3 4 5 6 7 8 9 10 11 12 13 14 3 4 5 6 7 8 9 10 11 12 13 14 ···
Y ··· 0 1 ···

FIGURE 3 Years 0 and 1 in the Gregorian/Julian and computational calendars

Gregorian/Julian calendar. Since the rules for the Julian and Gregorian calendars differ (Definitions 1 and 2), there are two corresponding rules on the computational calendar.

Definition 6 (Computational Julian rule). Year Y is a **computational Julian leap year** if $Y + 1$ is a multiple of 4.

Definition 7 (Computational Gregorian rule). Year Y is a **computational Gregorian leap year** if $Y + 1$ is a multiple of 4, except if $Y + 1$ is divisible by 100 but not by 400.

For each month $M \in \{3, \dots, 14\}$, Table 1 shows its name, number of days and accumulated number of days in prior months, that is, $m^*(M)$. As intended, m^* does not depend on whether the year is a leap year or common year: the number of days in February is irrelevant.

The following equations map a date $X = (Y, M, D)$ of the computational calendar into its corresponding date $X_{G/J} = (Y_{G/J}, M_{G/J}, D_{G/J})$ on the Gregorian/Julian calendar:

$$Y_{G/J} = Y + \mathbf{1}_{\{M \geq 13\}}, \qquad M_{G/J} = M - 12 \cdot \mathbf{1}_{\{M \geq 13\}}, \qquad D_{G/J} = D + 1, \tag{8}$$

where $\mathbf{1}_{\{M \geq 13\}}$ takes the value 1 if $M \geq 13$ and 0 otherwise. The reciprocal map is:

$$Y = Y_{G/J} + \mathbf{1}_{\{M_{G/J} \leq 2\}}, \qquad M = M_{G/J} + 12 \cdot \mathbf{1}_{\{M_{G/J} \leq 2\}}, \qquad D = D_{G/J} - 1, \tag{9}$$

where $\mathbf{1}_{\{M_{G/J} \leq 2\}}$ takes the value 1 if $M_{G/J} \leq 2$ and 0 otherwise.

Remark 1. The mathematically succinct expressions $\mathbf{1}_{\{M \geq 13\}}$ and $\mathbf{1}_{\{M_{G/J} \leq 2\}}$ indicate if the month is January or February, but there are alternative ways of determining this. For instance, Table 1 shows that the month is January or February if, and only if, $m^*(M) \geq 306$. Therefore, a date whose day of the year (in the computational calendar) is N_Y falls in January or February if, and only if, $N_Y \geq 306$.

Note, from $D_{G/J} = D + 1$, that we set the computational calendar to number days from zero, which is a convenient feature already seen in Section 2.

TABLE 1 Months of the computational calendar

M	Name	# days	$m^*(M)$
3	March	31	0
4	April	30	31
5	May	31	61
6	June	30	92
7	July	31	122
8	August	31	153
9	September	30	184
10	October	31	214
11	November	30	245
12	December	31	275
13	January	31	306
14	February	28 or 29	337

Algorithm 1. Finds the Julian date (Y_J, M_J, D_J) from its *rata die* N. (The epoch is (0, 3, 1) of the Julian calendar.)

$N_1 = 4 \cdot N + 3,$	$N_2 = 5 \cdot N_Y + 461,$	$J = \mathbf{1}_{\{M \geq 13\}},$
$Y = N_1/1461,$	$M = N_2/153,$	$Y_J = Y + J,$
$N_Y = N_1\%1461/4,$	$D = N_2\%153/5,$	$M_J = M - 12 \cdot J,$
		$D_J = D + 1.$

Table 1 shows that, except for $M = 14$, the number of days is periodic (the pattern $31 - 30 - 31 - 30 - 31$ repeats twice and starts again at $M = 13$) and there are 153 days in each 5-month period. This suggests the possibility that m^* is the EAF $m^*(M) = (153 \cdot M + b^*)/5$ for some b^*. By trial-and-error (a small program can perform this uninspiring task), we find that $b^* = -457$ works and m^* becomes:[##]

$$m^*(M) = (153 \cdot M - 457)/5, \quad \forall M \in [3, 14]. \tag{10}$$

This equation eliminates the need to store $m^*(M)$ in a look-up table.

Example 5 (Month EAF of the computational calendar). As seen in Example 4, we can find M and D from the day of the year N_Y by applying Theorem 1 to $f^* = m^*$. We have $a^* = 153$, $b^* = -457$ and $d^* = 5$, which yields $a = 5$, $b = 461$, $d = 153$, and

$$M = m(N_Y) = (5 \cdot N_Y + 461)/153 \qquad \text{and} \qquad D = m^\circ(N_Y) = (5 \cdot N_Y + 461)\%153/5.$$

5 ALGORITHMS FOR THE JULIAN CALENDAR

This section derives algorithms for the Julian calendar, building up arguments that will be used for the corresponding derivations for the Gregorian calendar in Section 6. To find a Julian date from its *rata die*, our algorithm first finds the corresponding date in the computational calendar and then maps the result to the Julian calendar. On the computational calendar, we follow the steps set out in Section 2 and Examples 2 and 4.

The epoch is set to $X_0 = (0, 3, 0)$, which is the first day of the year 0 of the computational calendar.

The derivation of y^* is similar to that of Equation (5), but must account for the correct leap year rule given by Definition 6. The number of computational leap years in $[0, Y[$ matches the number of multiples of 4 in $[1, Y + 1[= [1, Y]$, which is $Y/4$. Therefore, the number of days in all years in $[0, Y[$ is:

$$y^*(Y) = 365 \cdot Y + Y/4 = 1461 \cdot Y/4. \tag{11}$$

Example 6 (Year EAF for the computational Julian calendar). Let N be the *rata die* of $X = (Y, M, D)$. As seen in Example 2, we can find the year Y and day of the year N_Y from N by applying Theorem 1 to $f^* = y^*$. [Correction added on 21 December 2022, after first online publication: the Theorem number has been corrected in the preceding sentence.] We have $a^* = 1461$, $b^* = 0$, $d^* = 4$, which yields $a = 4$, $b = 3$, $d = 1461$, as well as:

$$Y = y(N) = (4 \cdot N + 3)/1461 \qquad \text{and} \qquad N_Y = y^\circ(N) = (4 \cdot N + 3)\%1461/4.$$

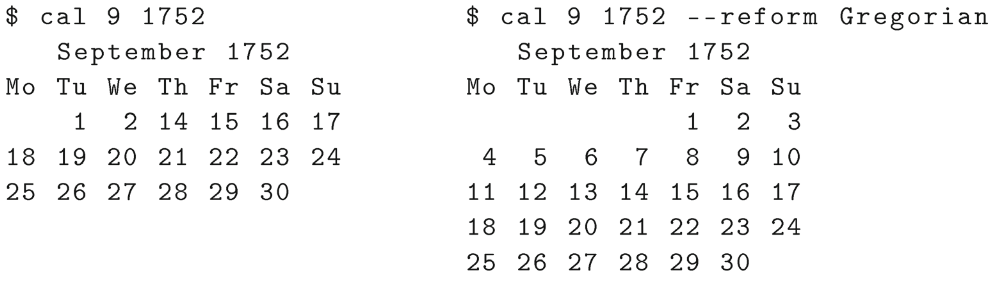
We have already found M and D from N_Y in Example 5, that is, $M = (5 \cdot N_Y + 461)/153$ and $D = (5 \cdot N_Y + 461)\%153/5$.

Putting everything together gives Algorithm 1. It finds the Julian date (Y_J, M_J, D_J) from its *rata die* N. Its first two columns work on the computational calendar and calculate the year Y, day of the year N_Y, month M and day D. The third column maps the date (Y, M, D) of the computational calendar to the date (Y_J, M_J, D_J) of the Julian calendar using Equation (8).

[##] Variations of Equation (10) have been found by many authors,[1,13,15-19] most often through trial-and-error. The periodicity observations above shed some light on this matter. However, more structured methods are available.[24] For instance, a simple textbook linear regression on the set $\{(M, m^*(M) + 0.5) \in \mathbb{Q}^2 \; ; \; M \in [3, 14]\}$ finds $m^*(M) = (26256 \cdot M - 78317)/858$. Actually, the final version of our algorithm will use neither. As we will see in Section 7, for a given EAF, a faster-to-evaluate alternative might exist. Using either of these two EAFs for m^* allows us to find the very same faster EAF that we will use.

Algorithm 2. Calculates the *rata die* N of the Julian date (Y_J, M_J, D_J) (The epoch is (0, 3, 1) of the Julian calendar)

```
$J = \mathbf{1}_{\{M_J \le 2\}}$,
$Y = Y_J - J$,
$M = M_J + 12 \cdot J$,
$D = D_J - 1$.

$y^* = 1461 \cdot Y/4$,
$m^* = (153 \cdot M - 457)/5$,
$N = y^* + m^* + D$.
```



```
$ cal 9 1752
   September 1752
Mo Tu We Th Fr Sa Su
    1  2 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

```
$ cal 9 1752 --reform Gregorian
   September 1752
Mo Tu We Th Fr Sa Su
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

FIGURE 4 On the left, the month of September 1752 in Britain and its colonies when they moved from the Julian calendar (up to the 2^{nd}) to the Gregorian (from the 14^{th}). On the right, the Gregorian calendar for the same month

Conversely, Algorithm 2 calculates the *rata die* N of a given Julian date (Y_J, M_J, D_J). The first column maps the given date to the date (Y, M, D) of the computational calendar. The second column calculates N using Equations (4), (10), and (11).

6 | ALGORITHMS FOR THE GREGORIAN CALENDAR

Although it is now used by almost every country in the world, the adoption of the Gregorian calendar was gradual. For instance, Britain and its colonies (including the US) only moved away from the Julian calendar in September 1752. Users of the *util-linux* software package can check this historical moment on the command line as seen on the left of Figure 4.

Eleven days from the 3rd to the 13th of September 1752 were skipped to synchronise the newly adopted calendar with other countries. British readers will consider the `cal` program historically accurate, but readers from another country might disagree. What everyone can agree on is that it is meaningless to refer to Gregorian dates prior to 1582, because this calendar did not exist before the reform of Pope Gregory XIII.

To avoid idiosyncrasies relating to adoption date, most implementations extrapolate the Gregorian calendar backwards and forwards indefinitely, yielding the so called **proleptic Gregorian calendar**.[1,3] The Linux utility `cal` also displays this calendar, as seen on the right of Figure 4. For the sake of brevity, we drop the adjective *proleptic* and refer to the Gregorian calendar even when dealing with dates prior to 1582.

As in Section 5, to find a Gregorian date from its *rata die*, our algorithm first finds the date in the computational calendar and then maps the result to the Gregorian calendar. The epoch is set to $X_0 = (0, 3, 0)$, which is the first day of year 0 of the computational calendar.

Unfortunately, the steps set out in Section 2 do not work here. Indeed, similarly to the derivation of Equation (11) but using the computational Gregorian rule (Definition 7), we can deduce that the number of computational leap years in $[0, Y[$ matches the number of years in $[1, Y]$ that are multiples of 4 except those that are divisible by 100 but not by 400, which is $Y/4 - Y/100 + Y/400$. Therefore, the number of days in all years in $[0, Y[$ is:

$$y^*(Y) = 365 \cdot Y + Y/4 - Y/100 + Y/400. \tag{12}$$

Sadly, this expression is *not* an EAF[‖‖] and we cannot apply Theorem 1.

[‖‖] For rational division we have

$$365 \cdot Y + \frac{Y}{4} - \frac{Y}{100} + \frac{Y}{400} = \frac{365 \cdot 400 \cdot Y + 100 \cdot Y - 4 \cdot Y + Y}{400} = \frac{146097 \cdot Y}{400}.$$

Not all hope is lost, however. For the Egyptian and Julian calendars, we proceeded in two steps related to year and month and applied Theorem 1 for y^* and m^*. For the Gregorian calendar, we simply need to add a step for the century, as we shall see now.

Let C be a century in the computational Gregorian calendar. From its first year, $100 \cdot C$, to its last, $100 \cdot C + 99$, there are $100/4 = 25$ values of Y such that $Y + 1$ is a multiple of 4: namely, $100 \cdot C + 3$, $100 \cdot C + 7$, $100 \cdot C + 11$, $\dots$, $100 \cdot C + 99$. For all these Ys except the last, $Y + 1$ is not a multiple of 100 and thus, from Definition 7, it is guaranteed to be a leap year. Hence, a century has at least 24 leap years and therefore at least 36524 days. However, a century might have 36525 days if $100 \cdot C + 99$ turns out to be a leap year. In this article, a century with 36525 days is called a **leap century**.

Now, for $Y = 100 \cdot C + 99$ we have $Y + 1 = 100 \cdot (C + 1)$ and this is a multiple of 400 if, and only if, $C + 1$ is a multiple of 4. Hence, the number of leap centuries in $[0, C[$ matches the number of multiples of 4 in $[1, C + 1[= [1, C]$, which is $C/4$. Therefore, the number of days in all centuries in $[0, C[$ is:

$$c^*(C) = 36524 \cdot C + C/4 = 146097 \cdot C/4. \tag{13}$$

Having previously seen that algebraic manipulations similar to the above cannot be applied to y^* to obtain an EAF, some readers might wonder if/why Equations (11) and (13) are correct. Accordingly, we provide a step-by-step proof of Equation (13):

$$\begin{aligned} 146097 \cdot C &= 146096 \cdot C + C \\ &= 146096 \cdot C + \left[4 \cdot (C/4) + C\%4\right] && \text{by Euclidean division of } C \text{ by } 4 \\ &= 4 \cdot \left[36524 \cdot C + C/4\right] + [C\%4] && \text{taking out common factor 4.} \end{aligned}$$

Since $0 \leq C\%4 < 4$, by Euclidean division, the last equality above means that the two terms inside the brackets are respectively the quotient and remainder of the division of $146097 \cdot C$ by 4. In particular, $146097 \cdot C/4 = 36524 \cdot C + C/4$, which is Equation (13). A similar argument holds for Equation (11).

Example 7 (Century EAF for the computational Gregorian calendar). Using Equation (13), we can derive expressions for the century $C = f(N)$ and the day of the century $N_C = f^\circ(N)$ from the *rata die* N. We use again the pattern that we already encountered (twice) on the Egyptian calendar. This is the same pattern we saw in abstract terms in Examples 2 and 4, which then became more concrete in Examples 5 and 6.

Let N be the *rata die* of $X = (Y, M, D)$. Theorem 1 applied to $f^* = c^*$, shows that $c(N) = (4 \cdot N + 3)/146097$ and $c^\circ(N) = (4 \cdot N + 3)\%146097/4$ respectively give the century $C = Y/100$ and the day of the century N_C.

We must now find $Z = Y\%100$, M and D from the day of the century N_C. Luckily, this is what Algorithm 1 does if we replace N with N_C and Y with Z. Indeed, by definition, the day of the century N_C is smaller than the number of days in century C. Hence, N_C does not cross a century boundary and, in this case, the Julian and Gregorian leap years match. The steps of Algorithm 1 can therefore be followed.

From the discussion above, we obtain Algorithm 3, which finds the Gregorian date (Y_G, M_G, D_G) from its *rata die* N. Its first three columns work on the computational calendar and calculate the century C, day of the century N_C, year of the century Z, day of the year N_Y, year Y, month M and day D. The fourth column maps the date (Y, M, D) of the computational calendar to the date (Y_G, M_G, D_G) of the Gregorian calendar using Equation (8).

Recall that Equation (12) is unfortunately not an EAF. It is correct though, and we shall use it in a slightly different form:

$$y^*(Y) = 1461 \cdot Y/4 - C + C/4, \quad \text{where} \quad C = Y/100. \tag{14}$$

Hence, although wrong, it is not irrational (forgive the pun) to expect that:

$$365 \cdot Y + Y/4 - Y/100 + Y/400 = 146097 \cdot Y/400.$$

A counter-example is $Y = 4$, for which the left side gives 1461 and the right side gives 1460.

Algorithm 3. Finds the Gregorian date (Y_G, M_G, D_G) from its *rata dieN* (The epoch is (0, 3, 1) of the Gregorian calendar)

```
$N_1 = 4 \cdot N + 3,$
$C = N_1/146097,$
$N_C = N_1\%146097/4,$

$N_2 = 4 \cdot N_C + 3,$
$Z = N_2/1461,$
$N_Y = N_2\%1461/4,$
$Y = 100 \cdot C + Z,$

$N_3 = 5 \cdot N_Y + 461,$
$M = N_3/153,$
$D = N_3\%153/5,$

$J = \mathbf{1}_{\{M \geq 13\}},$
$Y_G = Y + J,$
$M_G = M - 12 \cdot J,$
$D_G = D + 1.$
```

Algorithm 4. Calculates the *rata die* N of the Gregorian date (Y_G, M_G, D_G) (The epoch is (0, 3, 1) of the Gregorian calendar)

```
$J = \mathbf{1}_{\{M_G \leq 2\}},$
$Y = Y_G - J,$
$M = M_G + 12 \cdot J,$
$D = D_G - 1,$
$C = Y/100,$

$y^* = 1461 \cdot Y/4 - C + C/4,$
$m^* = (153 \cdot M - 457)/5,$
$N = y^* + m^* + D.$
```

The first term replaces $365 \cdot Y + Y/4$ (as seen in Equation (11)). Moreover, using $Y/400 = (Y/100)/4$ allows us to substitute the division by 400 with a division by 4, which is much cheaper.

Algorithm 4 calculates the *rata die* N of a given Gregorian date (Y_G, M_G, D_G). The first column maps the given date to the date (Y, M, D) of the computational calendar and also finds the century C. The second column calculates N using Eqs. (4), (10) and (14).

Algorithms 3 and 4 are purely arithmetic. They avoid loops and look-up tables and hence minimise branching and cache thrashing. They can be further optimised, however. These optimisations are the subject of next section.

7 | FAST EVALUATION OF EUCLIDEAN AFFINE FUNCTIONS

This section covers optimisations for $f(n) = (a \cdot n + b)/d$ and $f^{\circ}(n) = (a \cdot n + b)\%d/a$. For f, the particular case $a = 1$ and $b = 0$ has been considered by many authors,[4-8] who have derived strength reductions, including reducing n/d to a multiplication and cheaper operations. Major optimisers use the algorithms of Granlund and Montgomery.[6] Faster alternatives exist but compilers are not able to use them. For instance, n might be restricted to a small interval but optimisers, unaware of this fact, must assume that n can take any allowed value for its type, usually in an interval of form $[0, 2^w[$ or $[-2^{w-1}, 2^{w-1}[$. We change focus and instead of looking for an algorithm on a given interval, we start with the best algorithm we know and search the largest interval on which it can be applied. If such an interval is satisfactory for our application, we then use this algorithm.

Conceptually, to evaluate $(a \cdot n + b)/d$ we multiply n by $a \cdot d^{-1}$ and add the result to $b \cdot d^{-1}$. In this paragraph, we assume that $b = 0$. We take an integer approximation a' of $2^k \cdot a \cdot d^{-1}$, where $k \in \mathbb{Z}^+$ is carefully chosen, and evaluate $a' \cdot n/2^k$. If the approximation a' is good enough and n is not too large, so that the approximation error amplified by the multiplication by n is still small, then we might get $a \cdot n/d = a' \cdot n/2^k$. The latter expression has the advantage of having a power-of-two divisor and is thus much cheaper to evaluate. Two natural choices for a' are $\lceil 2^k \cdot a \cdot d^{-1} \rceil$ and $\lfloor 2^k \cdot a \cdot d^{-1} \rfloor$. In general, $2^k \cdot a \cdot d^{-1}$ is not an integer and thus, for positive operands, rounding up and down respectively gives $a' = 2^k \cdot a/d + 1$ and $a' = 2^k \cdot a/d$. Each of these two approximations is covered by one of the following theorems. (Their proofs are presented later in Section 14.1.)

Theorem 2 (Fast round-up EAF evaluation). *Let $k \in \mathbb{Z}^+$ and $f(n) = (a \cdot n + b)/d$ with $d > 0$. Set $a' = 2^k \cdot a/d + 1$, $b' = -\min\left\{a' \cdot n - 2^k \cdot f(n)\ ;\ n \in [0, d[\right\}$ and $\varepsilon = d - 2^k \cdot a\%d$. For $n \in [0, d[$ define*:

$$Q(n) = \min\left\{q \in \mathbb{Z}^+\ ;\ \varepsilon \cdot q \geq 2^k \cdot (1 + f(n)) - (a' \cdot n + b')\right\} \quad \text{and} \quad P(n) = d \cdot Q(n) + n.$$

Let $U = \min\{P(n)\ ;\ n \in [0, d[\ \}$. Then,

$$(a \cdot n + b)/d = \left(a' \cdot n + b'\right)/2^k, \quad \forall n \in [0, U[.$$

Proof. See Section 14.1 ■

Theorem 3 (Fast round-down EAF evaluation). *Let $k \in \mathbb{Z}^+$ and $f(n) = (a \cdot n + b)/d$ with $d > 0$ and $2^k \cdot a\%d > 0$. Set $a' = 2^k \cdot a/d$ and $b' = \min\left\{2^k - 1 - a' \cdot n + 2^k \cdot f(n)\ ;\ n \in [0, d[\right\}$ and $\varepsilon = 2^k \cdot a\%d$. For $n \in [0, d[$ define:*

$$Q(n) = \min\left\{q \in \mathbb{Z}^+\ ;\ \varepsilon \cdot q > (a' \cdot n + b') - 2^k \cdot f(n)\right\} \quad \text{and} \quad P(n) = d \cdot Q(n) + n.$$

Let $U = \min\{P(n)\ ;\ n \in [0, d[\ \}$. Then,

$$(a \cdot n + b)/d = (a' \cdot n + b')/2^k, \quad \forall n \in [0, U[.$$

Proof. See Section 14.1 ■

In both theorems above, a' and ε are obtained in $O(1)$ operations and b' is found by an $O(d)$ search.[***] $Q(n)$ is essentially obtained by a division by ε, which has $O(1)$ complexity, and consequently $P(n)$ also has $O(1)$ complexity. Hence, U and the overall search for a fast EAF have $O(d)$ time complexity. Since many EAFs featuring in calendrical calculations have small divisors, finding more efficient alternatives for them is reasonably fast.

The constants a', b' and the upper bound U depend on the choice of k and whether we use Theorem 2 or Theorem 3. As a comparison, the following examples show what the above theorems give for the same EAF and for the same k.

Example 8 (Fast round-up evaluation of m^* for the computational calendar). The number of days in months $[3, M[$ of the computational calendar is given by Equation (10): $m^*(M) = (153 \cdot M - 457)/5$. Theorem 2 applied to $k = 5$ and $f = m^*$, that is, for $a = 153$, $b = -457$ and $d = 5$, yields $a' = 980$, $b' = 2928$, $U = 12$ and

$$(153 \cdot M - 457)/5 = (980 \cdot M - 2928)/2^5, \quad \forall M \in [0, 12[. \tag{15}$$

Example 9 (Fast round-down evaluation of m^* for the computational calendar). The number of days in months $[3, M[$ of the computational calendar is given by Equation (10): $M = m^*(M) = (153 \cdot M - 457)/5$. Theorem 3 applied to $k = 5$ and $f = m^*$, that is, for $a = 153$, $b = -457$ and $d = 5$, yields $a' = 979$, $b' = -2919$, $U = 34$ and

$$(153 \cdot M - 457)/5 = (979 \cdot M - 2919)/2^5, \quad \forall M \in [0, 34[. \tag{16}$$

These examples give two fast alternatives for the same EAF. In this case, the latter has the advantage of being valid over a larger range, $[0, 34[$, as opposed to $[0, 12[$, but in other cases the opposite is true. In our application, M is a month of the computational calendar and belongs to the interval $[3, 14]$. Hence, the expression on the right side of Equation (15) is of no use to us since it does not give the expected result for $M \in \{12,\ 13,\ 14\}$ as calculated by the left side of the equation.

The choice of $k = 5$ is not totally arbitrary: it is the minimum value for which Theorem 3 yields a faster alternative evaluation for m^* on an interval that contains $[3, 14]$. Larger values of k can also be used, but a' is non-decreasing on k and we want to avoid it being too large. (More on that in Section 8.)

Example 10 (Fast round-down evaluation of m for the computational calendar). The month of the date whose day of the year is N_Y is given in Example 5: $M = m(N_Y) = (5 \cdot N_Y + 461)/153$. Theorem 3 applied to $k = 16$ and $f = m$, that is, for $a = 5$, $b = 461$ and $d = 153$ yields $a' = 2141$, $b' = 197913$, $U = 734$ and

$$(5 \cdot N_Y + 461)/153 = (2141 \cdot N_Y + 197913)/2^{16}, \quad \forall N_Y \in [0{,}734[. \tag{17}$$

The value $k = 16$ yields a range of validity $[0{,}734[$ containing $[0{,}365[$ – the range of the day of the year N_Y. Other values, even smaller than 16, are possible, but an explanation for taking $k = 16$ will be given in Section 8.

7.1 | Fast division–the special case $a = 1, b = 0$

This section examines n/d with $d > 0$, that is, the particular case of the EAF $f(n) = (a \cdot n + b)/d$ where $a = 1$, $b = 0$ and $d > 0$.

[***] The supplementary material webpage[25] contains C++ code that calculates the constants of Theorems 2 and 3, in addition to testing the validity of some expressions presented in this paper (e.g., Examples 8 to 10).

The next theorem appeared in Cavagnino and Werbrouck[5] but it can also be obtained as a corollary of Theorem 2 for $a = 1$ and $b = 0$. However, a more direct proof is presented in Section 14.2.

Theorem 4 (Fast division). *Let* $d, k \in \mathbb{Z}^+$ *with* $d > 0$. *Set* $a' = 2^k/d + 1, \varepsilon = d - 2^k\%d$ *and* $U = \lceil a' \cdot \varepsilon^{-1} \rceil \cdot d - 1$. *If* $\varepsilon \leq a'$, *then*:

$$n/d = a' \cdot n/2^k, \quad \forall n \in [0, U[.$$

Proof. See Section 14.2. ■

Example 11 (Fast division by 1461). Consider the division $n/1461$ that appears in Algorithm 1 (for $n = N_1$) and Algorithm 3 (for $n = N_2$). For $k = 32$, the constants given by Theorem 4 are $a' = 2939745$, $\varepsilon = 149$ and $U = 28825529$. Since $\varepsilon \leq a'$, this theorem gives:

$$n/1461 = 2939745 \cdot n/2^{32}, \quad \forall n \in [0, 28825529[. \tag{18}$$

The choice of $k = 32$ will be explained in Section 8.

7.2 | Fast evaluation of residual functions

Theorems 2 and 3 provide optimisations for evaluating EAFs, generalising the current practice for divisions revisited by Theorem 4. They can be used for an EAF f and for its minimal right inverse f^*. The next result extends the optimisation to residual functions and, in essence, states that if f' matches f in a given interval (e.g., f' might be a faster alternative for f), then $f^{\circ\prime}$ also matches f° in the same interval.

Theorem 5 (Alternative residual evaluation). *Let* $f(n) = (a \cdot n + b)/d$ *and* $f'(n) = (a' \cdot n + b')/d'$, *with* $d \geq a > 0$, *and assume that* $f(n) = f'(n)$ *for all* $n \in [L, U[$. *If* $f^*(f(L)) = L$ *and* $f'(L-1) < f'(L)$, *then*:

$$f^\circ(n) = f^{\circ\prime}(n) \quad \forall n \in [L, U[.$$

Proof. See Section 14.3. ■

In the case where $f'(n) = (a' \cdot n + b')/2^k$, we have $f^{\circ\prime}(n) = (a' \cdot n + b')\%2^k/a'$, so that the calculation of the remainder is cheap. Although this is the most interesting application for us, the theorem also applies for other divisors. Hence, the word *Alternative* instead of *Fast* in the theorem's name.

Example 12 (Fast remainder of the division by 1461). Consider the EAFs $f(n) = n/1461$, $f^*(q) = q \cdot 1461$ and $f'(n) = 2939745 \cdot n/2^{32}$. Equation (18) shows that $f(n) = f'(n)$ for all $n \in [0, 28825529[$. Simple calculations give $f^*(f(0)) = 0$ and $f'(-1) = -1 < 0 = f'(0)$. Hence, it follows from Theorem 5 that:

$$n\%1461 = 2939745 \cdot n\%2^{32}/2939745, \quad \forall n \in [0, 28825529[. \tag{19}$$

Example 13 (Fast calculation of day for the computational calendar). Example 5 shows that if the day of the year of $X = (Y, M, D)$ is N_Y, then $D = m^\circ(N_Y) = (5 \cdot N_Y + 461)\%153/5$ and m° is the residual function of $m(N_Y) = (5 \cdot N_Y + 461)/153$ whose minimum right inverse is $m^*(M) = (153 \cdot M - 457)/5$. In light of Equation (17), we wish to apply Theorem 5 to $f = m, f'(N_Y) = (2141 \cdot N_Y + 197913)/2^{16}$, $L = 0$ and $U = 734$. For this, we need to verify that $m^*(m(0)) = 0$ and that $f'(-1) < f'(0)$. Now, $m(0) = (5 \cdot 0 + 461)/153 = 461/153 = 3$ and then, $m^*(m(0)) = m^*(3) = (153 \cdot 3 - 457)/5 = (459 - 457)/5 = 2/5 = 0$. We also have $f'(-1) = (-2141 + 197913)/2^{16} = 195772/65536 = 2$ and $f'(0) = 197913/2^{16} = 197913/65536 = 3$, so that $f'(-1) < f'(0)$. Theorem 5 therefore gives:

$$(5 \cdot N_Y + 461)\%153/5 = (2141 \cdot N_Y + 197913)\%2^{16}/2141, \quad \forall N_Y \in [0,734[. \tag{20}$$

8 | OPTIMISED ALGORITHMS FOR THE GREGORIAN CALENDAR

The remainder of our article focuses on algorithms for the Gregorian calendar. Of course, similar (and often simpler) arguments apply to the Julian calendar. This section explains Algorithms 5 and 6, which are, respectively, optimised versions of Algorithms 3 and 4.

Algorithm 5. Finds the Gregorian date (Y_G, M_G, D_G) from its *rata dieN* (The epoch is (0, 3, 1) of the Gregorian calendar)

$N_1 = 4 \cdot N + 3,$ $C = N_1/146097,$ $N_C = N_1\%146097/4,$	$N_2 = 4 \cdot N_C + 3,$ $P_2 = 2939745 \cdot N_2,$ $Z = P_2/2^{32},$ $N_Y = P_2\%2^{32}/2939745/4,$ $Y = 100 \cdot C + Z,$	$N_3 = 2141 \cdot N_Y + 197913,$ $M = N_3/2^{16},$ $D = N_3\%2^{16}/2141,$	$J = \mathbf{1}_{\{N_Y \geq 306\}}$ $Y_G = Y + J,$ $M_G = M - 12 \cdot J,$ $D_G = D + 1.$

Algorithm 6. Calculates the *rata die* N of the Gregorian date (Y_G, M_G, D_G) (The epoch is (0, 3, 1) of the Gregorian calendar)

$Y = Y_G - \mathbf{1}_{\{M_G \leq 2\}},$ $M = M_G + 12 \cdot \mathbf{1}_{\{M_G \leq 2\}},$ $D = D_G - 1,$ $C = Y/100,$	$y^* = 1461 \cdot Y/4 - C + C/4,$ $m^* = (979 \cdot M - 2919)/2^5,$ $N = y^* + m^* + D.$

Addition to $(2^{31} - 1)$:

```
add rdi, 2147483647
```

Addition to $(2^{31} + 1)$:

```
mov eax, 2147483649
add rax, rdi
```

FIGURE 5 Assembly emitted by Clang 15.0.0 (with `-O3`) for the addition to constants $2^{31} - 1$ and $2^{31} + 1$.

The only difference between Algorithms 4 and 6 is the calculation of m^*. The validity of the transformation was seen in Example 9, and performance is improved by replacing division by 5 with division by 2^5.

Similarly, Algorithm 5 is largely obtained from the results of Section 7 applied to the EAFs seen in Algorithm 3. However, there is much more behind it than just using divisors that are powers of two. The sequel covers other aspects related to modern superscalar CPUs and, more specifically, those of the x86_64 family. The presentation is split into subsections, each of which compares a particular column of Algorithms 3 and 5.

8.1 | First column

Algorithms 3 and 5 share the first column, which begs the question, Why do we not apply the results of Section 7 to find faster alternatives for $(4 \cdot N + 3)/146097$ and $(4 \cdot N + 3)\%146097/4$. Recall that such transformations are valid only on a bounded interval, and since there are no limits on the rata die N this cannot be done. Well, this is the theory but, in practice, applications set limits on the range of dates they support, which in turn set bounds on N. With this information in hand, the results of Section 7 *might* be applied. Other aspects must nonetheless be considered: the exponent k in Theorems 2 and 3 must be large enough for U to surpass the requirement on N. The larger k becomes, the larger a' is. If a' gets too large, then x86_64 cannot use a' as an immediate value of arithmetic instructions and it must instead be loaded into a register first, and there might be a price to pay for that. To illustrate this point, Figure 5 provides the assembly code for the addition to two different constants. Note the extra `mov` required for the addition to the larger constant.

Although the first column does not profit from the results of Section 7, it uses Theorem 1 for $f^\circ = c^\circ$ to obtain the expression for N_C given by Equation (7). This allows for some interesting bit twiddling: the calculation of N_C requires that of $R = N_1\%146097$, from which N_2 is obtained by $N_2 = 4 \cdot (R/4) + 3$. It is easy to see that this expression is equivalent to $R \mid 3$, where $\mid$ denotes bitwise-`or`. Hence, the calculations of N_C and N_2 can be replaced by:

$$R = N_1\%146097 \qquad \text{and} \qquad N_2 = R \mid 3.$$

In contrast, unaware of Equation (7), some implementations[17,18] use the alternative $N_C = N - c^*(c(N)) = N - 146097 \cdot C/4$, whose last operation is the subtraction and not the division by 4 necessary for this bit twiddling. Figure 6 contrasts

$$N_1 = 4 \cdot N + 3,$$
$$C = N_1/146097,$$
$$N_C = N_1\%146097/4,$$
$$N_2 = 4 \cdot N_1 + 3.$$

```
lea   eax, [4*rdi + 3]
imul  rcx, rax, 963315389
shr   rcx, 47
imul  ecx, ecx, 146097
sub   eax, ecx
or    eax, 3
```

$$N_1 = 4 \cdot N + 3,$$
$$C = N_1/146097,$$
$$N_C = N - 146097 \cdot C/4,$$
$$N_2 = 4 \cdot N_C + 3.$$

```
lea   eax, [4*rdi + 3]
imul  rax, rax, 963315389
shr   rax, 47
imul  eax, eax, 146097
shr   eax, 2
sub   edi, eax
lea   eax, [4*rdi + 3]
```

FIGURE 6 Assembly emitted by Clang 15.0.0 (with `-O3`) for two alternatives up to the calculation of N_2

the assembly for the two alternatives. As we can see, the code on the left replaces two instructions seen on the right, namely, `shr` (3 bytes) and `lea` (7 bytes), with a single `or` (3 bytes). The code on the left has fewer instructions and is 7 bytes shorter.

8.2 | Second column

The input of the second column of Algorithms 3 and 5 is $N_C = N_1\%146097/4$ and we have $0 \le N_C \le 146096/4 = 36{,}524$. Algorithm 3 evaluates $Z = (4 \cdot N_C + 3)/1461$ and $N_Y = (4 \cdot N_C + 3)\%1461/4$, and the results of Section 7 can be used to find faster alternatives valid for $N_C \in [0, 36525[$. However, doing so would forbid the bitwise-`or` trick explained above as it relies on the expression $4 \cdot N_C + 3$. Hence, we keep the numerator N_2 as in Algorithm 3 and seek to optimise the calculations of the quotient $Z = N_2/1461$ and remainder $N_Y = N_2\%1461$. Now, since $N_C \in [0, 36525[$, we have $0 \le N_2 \le 4 \cdot 36{,}524 + 3 = 146099$. Therefore, $N_2 \in [0, 28825529[$ and Equations (18) and (19) are used in Algorithm 5.

The second column of Algorithm 5 might look worse than that of Algorithm 3 but the opposite is true:[†††] firstly, the divisions by 2939745 and 4 can be collapsed into a single division by $2939745 \cdot 4 = 11758980$ and, secondly, following the results of Granlund and Montgomery,[6] compilers use the result of Theorem 4 for $k = 54$ and apply this strength reduction:

$$n/11758980 = 1531969483 \cdot n/2^{54}, \quad \forall n \in [0, 10441974239[.$$

Similarly, for Algorithm 3, compilers take $k = 39$ and Theorem 4 gives:

$$Z = N_2/1461 = 376287347 \cdot N_2/2^{39}, \quad \forall N_2 \in [0, 6958934390[,$$

and for the remainder, they use $N_2\%1461 = N_2 - 1461 \cdot (N_2/1461)$. In summary, they calculate:

$$Z = 376287347 \cdot N_2/2^{39} \qquad \text{and} \qquad N_Y = (N_2 - 1461 \cdot Z)/4.$$

Note the dependency of N_Y on Z in the expressions above. This forces the CPU to wait for the calculation of Z to finish before the calculation of N_Y can start. We emphasise that this derives from the dependency of the remainder $N_2\%1461$ on the quotient $N_2/1461$. In contrast, there is no dependency of $P_2\%2^{32}$ (which simply resets all but the 32 lower bits of P_2) on $P_2/2^{32}$ or of N_Y on Z as calculated in Algorithm 5. Hence, once P_2 is obtained, the evaluations of Z and N_Y can start concurrently. Algorithm 5 can therefore benefit from the instruction-level parallelism of modern superscalar CPUs.

[†††] We rely on the transformations performed by some compilers. However, we can always manually enforce these transformations directly in the source code, as we will explain further on.

There is a small but worthwhile price to pay for this, which, due to our choice of $k = 32$, can be avoided in x86_64 CPUs as we will see now.

The evaluations of Z and N_Y of Algorithm 5 need P_2 to be stored in two different registers, which requires making a copy (`mov`) of P_2. This does not necessarily increase the number of instructions. Indeed, for backward compatibility with older 32-bit CPUs, x86_64, the `mov` instruction might copy the lower 32-bits of one register into another while resetting higher bits in the destination. Since this resetting is exactly what $P_2\%2^{32}$ does, this operation becomes unnecessary.

Figure 7 contrasts the assembly for the two alternative evaluations of Z and N_Y and presents a timeline analysis of execution. Each line corresponds to an instruction and contains two important marks: the first `e` indicates the start of the execution and `E` indicates its end. On the right, execution is sequential, with the start of any instruction only happening at the end of the previous one. On the left, once the calculation of P_2 (line 1) is finished, those of Z (line 3) and N_Y (lines 2, 4–5) begin. Consequently, the code on the left takes 10 cycles (line 0 shows cycle numbers modulus 10), whereas the code on the right takes 12.

8.3 | Third column

The third column of Algorithm 5 comes from Algorithm 3 and Equations (17) and (20). The assembly for the two alternatives and their respective timelines are shown in Figure 8. Most arguments seen above apply here, but we briefly touch on a few points.

$$P_2 = 2939745 \cdot N_2$$
$$Z = P_2/2^{32},$$
$$N_Y = P_2\%2^{32}/2939745/4.$$

```
0 0123456789
1 DeeeER   . imul  rax, rdi, 2939745
2 D===ER   . mov   ecx, eax
3 D===eER  . shr   rax, 32
4 D===eeeER. imul  rcx, rcx, 1531969483
5 D======eER shr   rcx, 54
```

$$Z = N_2/1461,$$
$$N_Y = N_2\%1461/4.$$

```
0 012345678901
1 DER  .    .. mov   eax, edi
2 DeeeER    .. imul  rax, rax, 376287347
3 D===eER   .. shr   rax, 39
4 D====eeeER.. imul  ecx, eax, 1461
5 D=======eER. sub   edi, ecx
6 D========eER shr   edi, 2
```

FIGURE 7 Assembly emitted by Clang 15.0.0 (with `-O3`) for the two alternatives for Z and N_Y with their respective timelines produced by llvm-mca (with `-timeline -iterations=1 -march=x86-64 -mcpu=alderlake`)

$$N_3 = 2141 \cdot N_Y + 197913$$
$$M = N_3/2^{16},$$
$$D = N_3\%2^{16}/2141.$$

```
0 012345678901
1 DeeeER    .. imul   eax, edi, 2141
2 D===eER   .. add    eax, 197913
3 D====eER  .. movzx  ecx, ax
4 D====eER  .. shr    eax, 16
5 D=====eeeER. imul   ecx, ecx, 31345
6 D========eER shr    ecx, 26
```

$$N_3 = 5 \cdot N_Y + 461$$
$$M = N_3/153,$$
$$D = N_3\%153/5.$$

```
0  012345678901234567
1  DeER .    .    . . lea    eax, [rdi + 4*rdi]
2  D=eER.    .    . . add    eax, 461
3  DeE-R.    .    . . mov    ecx, 3593175255
4  D==eeeER  .    . . imul   rcx, rax
5  D=====eER .    . . shr    rcx, 39
6  D======eeeER   . . imul   edx, ecx, 153
7  .D========eER  . . sub    eax, edx
8  .D=========eeeER . imul   eax, eax, 205
9  .D============eER. movzx  eax, ax
10 .D=============eER shr    eax, 10
```

FIGURE 8 Assembly emitted by Clang 15.0.0 (with `-O3`) for the two alternatives for M and D with their respective timeline produced by llvm-mca (with `-timeline -iterations=1 -march=x86-64 -mcpu=alderlake`)

$$N_3 = 2141 \cdot N_Y + 197913$$
$$M = N_3/2^{16},$$
$$J = \mathbf{1}_{\{N_Y \geq 306\}}.$$

```
0 01234567
1 DeeeER .  imul   eax, edi, 2141
2 D===eER.  add    eax, 197913
3 D====eER  shr    eax, 16
4 DeeE---R  xor    ecx, ecx
5 DeE----R  cmp    edi, 306
6 .DeeE--R  setae  cl
```

$$N_3 = 5 \cdot N_Y + 461$$
$$M = N_3/153,$$
$$J = \mathbf{1}_{\{M \geq 13\}}.$$

```
0 012345678
1 DeER .  . lea    eax, [rdi + 4*rdi]
2 D=eER.  . add    eax, 461
3 DeE-R.  . mov    ecx, 3593175255
4 D==eeeER. imul   rcx, rax
5 D=====eER shr    rcx, 39
6 DeeE----R xor    edx, edx
7 .D=eE---R cmp    eax, 1989
8 .D==eeE-R setae  dl
```

FIGURE 9 Assembly emitted by Clang 15.0.0 (with `-O3`) for J with their respective timeline produced by llvm-mca (with `-timeline -iterations=1 -march=x86-64 -mcpu=alderlake`)

In Algorithm 3 (right of Figure 8), the division by 153 is strength-reduced and uses a multiplication by 3593175255. This suffers from the issue illustrated by Figure 5: the multiplier is too large and cannot be an intermediate value operand of `imul`, and must be first loaded into a register (line 3).[‡‡‡]

The execution depicted by the right timeline is essentially sequential, with almost all instructions starting only when the previous one has finished. In contrast, the left timeline shows that the calculations of M (line 4) and D (lines 3, 5–6) start concurrently.

Finally, our choice of exponent 16 is also special for x86_64 CPUs. It allows the compiler to use `movzx ecx, ax` (line 3), which copies 16 bits from `ax` to `ecx` and resets the upper 16-bits of the destination, rendering unnecessary the operation %16.

8.4 | Fourth column

Remark 1 presents two alternative expressions for J, namely, $\mathbf{1}_{\{M \geq 13\}}$ and $\mathbf{1}_{\{N_Y \geq 306\}}$. They are used in Algorithms 3 and 5, respectively. The second expression for J, $\mathbf{1}_{\{N_Y \geq 306\}}$, has the advantage that N_Y is obtained earlier than M, allowing the CPU to start J's evaluation at earlier stages and while it performs other calculations in parallel.

Figure 9 shows assembly and timelines for the relevant parts of Algorithm 5 (left) and Algorithm 3 (right). Both start at the point just after N_Y is obtained.

The code on the right offers a pleasant surprise: since $M = N_3/153$, we deduce that $M \geq 13$ if, and only if, $N_3 \geq 13 \cdot 153 = 1989$, which is the test seen in the assembly. This eliminates the dependency of J on M and the need for the CPU to wait for the result of M to start the calculation of J. Still, in J's evaluation (lines 6–8), the comparison with 1989 only starts *after* the calculation of N_3 (lines 1–2) has finished. In contrast, for the code on the left, J's evaluation (lines 4–6) starts *at the same time* as N_3's (lines 1–2). In total, the snippet on the left takes 8 cycles and that on the right takes 9.

9 | CHANGING THE EPOCH

Our algorithms so far have set the epoch to $X_0 = (0, 3, 1)$, that is, 1 March 0000, and this section explains how they can be adapted to another epoch X_0'. Let K be the number of days from X_0 to X_0', that is, K is the output of Algorithm 4[§§§] for $X = X_0'$.

[‡‡‡] Again, the compiler does not know that $N_Y \in [0{,}365]$ so that $N_3 \in [461, 2286[$, and pessimistically assumes that N_3 can take any possible unsigned 32-bit value, implying the large multiplier.

[§§§] Performance is not a concern of this section and everything said about Algorithm 4 (resp., Algorithm 3) applies equally to Algorithm 6 (resp., Algorithm 5).

Let N' be a given number of days from X_0' to an unknown date X. The number of days from X_0 to X is then $N = K + N'$. Hence, using the latter as an input of Algorithm 3 recovers X. Reciprocally, given a date X, Algorithm 4 yields the number N of days from X_0 to X. $N' = N - K$ is then the number of days from X_0' to X.

The case where $K = 146097 \cdot s$, for some $s \in \mathbb{Z}$, is quite interesting. To assess the impact of a shift by this amount, let N be a given *rata die*, let N_1, C and N_C be as calculated by Algorithm 3 and let N_1', C' and N_C' be the corresponding quantities for the shifted *rata die* $N' = N + 146097 \cdot s$. It is easy to see that:

$$N_1' = N + 4 \cdot 146097 \cdot s, \qquad C' = C + 4 \cdot s \qquad \text{and} \qquad N_C' = N_C.$$

N_C is hence invariant in this shift and so are all other quantities that only depend on N_C. In particular, M_G and D_G are invariant. The century, however, shifts by $4 \cdot s$ which means that Y and Y_G shift by $400 \cdot s$. In summary, shifting *rata die* values by $146097 \cdot s$ is equivalent to shifting years by $400 \cdot s$.

10 | CHOOSING THE RIGHT TYPES

Performances of Algorithms 3 and 5 are deeply affected by types, since divisions of unsigned integer types are usually faster than their signed counterparts. It is a common misconception that division by 2 is simply a bit shift to the right. However, as Figure 10 shows, this does not hold for signed division in x86_64. Similar differences (extra shift and addition) are seen for all divisors.

It gets worse! For negative dividends, Euclidean division (which our algorithms rely on) does not match the integer division prescribed by the C and C++ Standards. A workaround applied by some implementations[11,18] is a conditional adjustment similar to Figure 11. It is possible to avoid such adjustments and operate mainly on unsigned integers since most of the quantities in our algorithms are non-negative numbers. For instance, in Algorithms 3 and 5, we have $N_C = R/4$, where $R = N_1 \% 146097$, and since remainders are always non-negative, we get $N_C \geq 0$. A quick inspection reveals that the only quantities that might be negative are N, N_1, C, Y in Algorithms 3 and 5, and Y, C and y^* in Algorithms 4 and 6. All other variables should have unsigned types.

The epoch and application constraints also play a role in the choice of types. The basic forms of our algorithms use the epoch 1 March 0000. If this is adequate and the application only supports dates on or after this epoch, then *rata die* values are non-negative and a quick look at the algorithms shows that all the quantities they manipulate are positive numbers and, thus, all variables should have unsigned types.

Upper bounds are equally important for the choice of types. In Section 8.2, we found that $N_C \leq 36524$, from which it follows that $N_2 \leq 146099$. Similar arguments give bounds for all quantities of Algorithms 3 and 5 as summarised below:

$$\begin{array}{cccc}
N_C \in [0, 36564], & Z \in [0, 99], & M \in [3, 14], & M_G \in [1, 12], \\
N_2 \in [0, 146099], & N_Y \in [0,365], & D \in [0, 30], & D_G \in [1, 31]. \\
P_2 \in [0, 429493804755], & N_3 \in [0, 979378], & J \in [0, 1], &
\end{array}$$

P_2 therefore fits in 64-bits and all the other variables fit in 32-bit unsigned integers.

Unsigned division by 2:

```
shr edi
```

Signed division by 2:

```
mov eax, edi
shr eax, 31
add eax, edi
sar eax
```

FIGURE 10 Assembly emitted by Clang 15.0.0 (with `-O3`) for unsigned (left) and signed (right) division by 2

```
int32_t Euclidean_quotient(int32_t const n, uint32_t const d) {
  return n >= 0 ? n / d : (n - (((int32_t) d) - 1)) / ((int32_t) d);
}
```

FIGURE 11 Obtaining the Euclidean quotient from C/C++ integer division

TABLE 2 For each relevant date, N is its *rata die* (with respect to $X_0 = (0, 3, 1)$) and N' is the value of N shifted by 11979954.

Date	N	N'
1 January −32767	−11967960	11994
1 January 1970	719468	12699422
31 December 32767	11968205	23948159

11 | ALGORITHMS FOR THE GREGORIAN CALENDAR WITH CUSTOMISED EPOCHS

This section uses the results of the previous two to obtain a concrete, highly efficient C/C++ implementation of the algorithms for the Gregorian calendar with a customised epoch.

To be more concrete, we will consider the Unix epoch 1 January 1970, but the same arguments apply to other dates. Similarly, we consider the requirements that the C++ Standard imposes on implementations: they must support all dates from 1 January −32767 to 31 December 32767.

The *rata die* with respect to $X_0 = (0, 3, 1)$ for each of the relevant dates (calculated by Algorithm 4) is presented in Table 2. The range for which the implementation is required to be correct is thus $[-11967960, 11968205]$. This interval contains negative numbers, which, *in principle*, requires us to evaluate Euclidean divisions on signed integers. As seen in Section 10, this degrades performance. However, from Section 9, we know that the months and days obtained by Algorithms 3 and 5 are invariant with respect to *rata die* shifts of the form $146097 \cdot s$, whereas years move by $400 \cdot s$. Therefore, if s is large enough, the corresponding shift in *rata die* values will bring them to positive territory without affecting months and days. Moreover, year moves can be corrected by subtracting $L = 400 \cdot s$. This allows us to work on unsigned integers throughout, except for the *rata die* shift and year correction, but these are additions and subtractions and do not suffer from the performance issue that affects the division of signed types.

For instance, for $s = 82$ we have $-11967960 + 146097 \cdot s = -11967960 + 11979954 = 11994 > 0$. The right column of Table 2 shows each value of $N' = N + 11979954$. It also shows that moving the epoch to 1 January 1970 requires an extra shift of 719468. Hence, the total shift is $719468 + 146097 \cdot s$, which amounts to $K = 12699422$ when $s = 82$.

Shifts for $s \geq 82$ also work. The choice of s obviously plays a role on the range of dates for which the implementation produces correct results. This range is also bounded above by the possibility of overflow. Each unit increase in s moves back the minimum and maximum supported dates by 400 years.[¶¶¶]

Putting these arguments together with Algorithm 5 yields the complete C++ implementation shown in Figure 12. We refrained from applying some previously mentioned optimisations that certain compilers might figure out by themselves (e.g., the bitwise-`or` trick), but implementers are encouraged to look at the assembly code generated by their compilers and decide whether they need/want to manually perform these optimisations.

Since Algorithm 6 is the inverse of Algorithm 5, their required shift and correction are in the opposite direction and reversed order. Figure 13 shows a complete C++ implementation that applies this correction and shift to Algorithm 6.

12 | PERFORMANCE ANALYSIS

We benchmarked our implementations against counterparts in five of the most widely used C, C++, C#, and Java libraries, as listed below:

glibc[11,20]	The GNU C Library.
Boost[17]	The Boost C++ libraries.
libc++[18]	LLVM's implementation of the C++ Standard Library.
.NET[12]	Microsoft .NET framework.
OpenJDK[21]	Oracle's open source implementation of the Java Platform SE (Android[22] uses the same code).

We used source files as publicly available on 2 May 2020. Non-C++ implementations have been ported to this language and have all been slightly modified to achieve consistent (a) function signatures; (b) storage types (for year, month, day

[¶¶¶] Our implementation in GCC sets $s = 3670$ to move the middle of the interval of validity closer to the epoch. The implementation is valid in a range far greater than the C++ Standard requirement.

TABLE 3 Relative CPU times for several platforms. The baseline (time = 1) is the code of Figure 12

Platform	.NET	Baum	Boost	Fliegel-Flandern	glibc	Hatcher	libc++	OpenJDK	Reingold-Dershowitz
clang_11.0.0-linux-intel_i7_10510U	4.23	1.51	1.41	2.69	8.14	2.83	2.27	2.43	7.51
clang_11.0.0-linux-ryzen_7_1800X	3.78	1.57	1.43	2.79	8.02	2.90	2.40	2.68	7.48
clang_12.0.0-windows-ryzen_7_1800X	4.12	2.01	1.41	2.99	8.27	2.78	2.47	2.35	8.46
clang_14.0.6-linux-intel_i7_10510U	5.19	2.20	1.60	3.22	10.24	2.93	2.65	2.80	9.56
clang_14.0.6-linux-ryzen_7_1800	4.26	2.04	1.49	3.01	8.47	2.76	2.51	2.91	8.80
gcc_10.2.0-linux-intel_i7_10510U	4.28	1.54	1.31	2.41	7.33	2.45	2.20	2.18	7.81
gcc_10.2.0-linux-ryzen_7_1800X	3.65	1.50	1.23	2.37	6.67	2.31	2.23	2.12	7.40
gcc_12.1.0-linux-intel_i7_10510U	4.24	1.58	1.25	2.47	7.67	2.53	2.29	2.37	7.99
gcc_12.1.0-linux-ryzen_7_1800X	3.65	1.69	1.23	2.48	7.08	2.42	2.26	2.31	7.67
msvc_19.29-windows-ryzen_7_1800X	3.85	1.88	1.27	2.31	7.12	2.12	2.49	2.31	7.81
msvc_19.29-windows-intel_i7_8750H	3.86	1.83	1.46	2.44	7.42	2.19	2.62	2.37	8.23

and *rata die*); and (c) epoch (1 January 1970). Some originals deal with date and time but our variants work on dates only. (Given the uniform durations of days, hours, minutes and seconds, it would be trivial to incorporate the time component to any dates-only algorithm.)

We did not include Microsoft's C++ Standard Library because back in May 2020 it did not yet implement these functionalities. Two other notable absences are libstdc++, the GNU implementation of the C++ Standard Library, and the Linux Kernel because we have contributed our algorithms to those systems. They are available in libstdc++ from the release of version 11 of the GNU Compiler Collection (GCC).[26] The Linux Kernel[27,28] features versions of Algorithm 5 from the release of version 5.14. Finally, the version of .NET that we used is outdated since they have recently replaced most of their old implementation with our algorithm.

We also considered our own implementations of algorithms described in the academic literature, namely Baum,[13] Fliegel and Flandern,[14] Hatcher[1,15,16] and Reingold and Dershovitz.[19] The code of all implementations and the build instructions are available on github[25]

Our time measurements were obtained with the help of the Google Benchmark library,[29] to which we delegated the task of producing statistically relevant results. The data shown in Figs. 14 and 15 were obtained on Windows 10 running on Intel i7 8750H at 2.22 GHz. The code was compiled by MSVC version 19.29 at optimisation level `/O2`.

The table in Figure 14 shows the time taken by each implementation to find the date for 16384 pseudo-random *rata die* values, uniformly distributed in $[-146097, 146097[$ (i.e., Unix epoch ±400 years). They encompass the time spent scanning the array of values (also shown). Subtracting the scanning time from that of each implementation gives a fairer account of the time spent by the algorithm itself. The chart plots these adjusted timings relative to ours (code of Figure 12).

Similarly, the table in Figure 15 shows the time taken by each algorithm to find *rata die* values corresponding to 16384 dates, uniformly distributed in $[(1570, 1, 1), (2370, 1, 1)[$ (again, Unix epoch ±400 years). The chart displays adjusted times relative to ours (code of Figure 13).

The Tables 3 and 4 show the values described above but for several platforms.

13 | PROOFS OF RESULTS OF SECTION 3

The remaining sections of this article present rigorous mathematical proofs for the results we have used to derive our algorithms. This section covers the algebraic results of Section 3 and Section 14 sets out the numerical approximations presented in Section 7.

It is worth recalling the definition of Euclidean division: given integers n and d, with $d \neq 0$, there exist *unique* integers q and r such that $0 \leq r < |d|$ and $n = q \cdot q + r$. They are denoted $q = n/d$ and $r = n\%d$ and are respectively called the **quotient** and the remainder of the division of n by d.

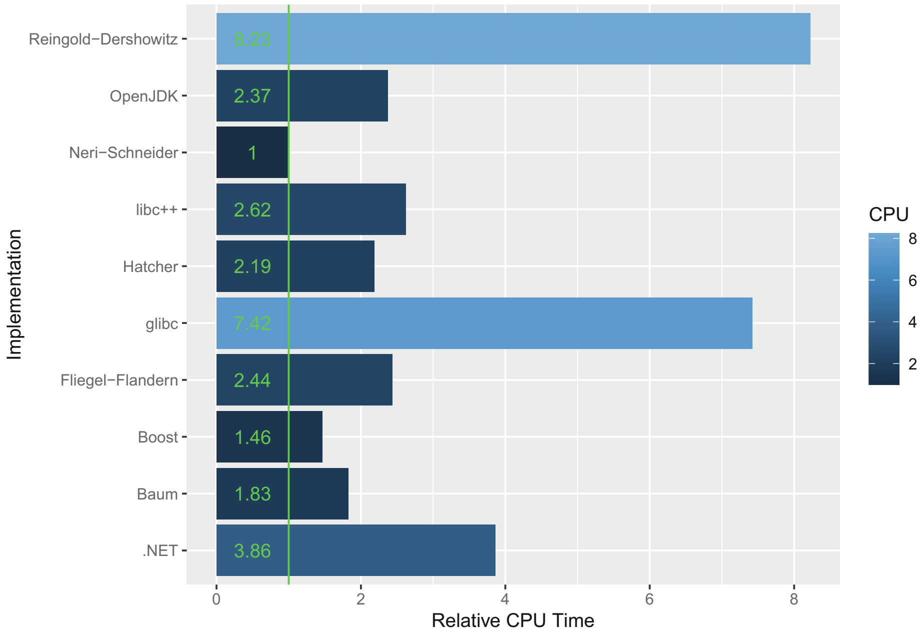
Note the emphasis on the word *unique* above: there are many ways of decomposing n as $n = d \cdot q + r$ (e.g., for $n = 17$ and $d = 5$ we have $17 = 3 \cdot 5 + 2$ and $17 = 2 \cdot 5 + 7$) and it is therefore wrong to directly deduce from such a decomposition

```
struct date32_t { int32_t year; uint32_t month; uint32_t day; };

date32_t to_date(int32_t N_U) {

  // Shift and correction constants.
  static uint32_t const s = 82;
  static uint32_t const K = 719468 + 146097 * s;
  static uint32_t const L = 400 * s;

  // Rata die shift.
  uint32_t const N = N_U + K;

  // Century.
  uint32_t const N_1 = 4 * N + 3;
  uint32_t const C   = N_1 / 146097;
  uint32_t const N_C = N_1 % 146097 / 4;

  // Year.
  uint32_t const N_2 = 4 * N_C + 3;
  uint64_t const P_2 = uint64_t(2939745) * N_2;
  uint32_t const Z   = uint32_t(P_2 / 4294967296);
  uint32_t const N_Y = uint32_t(P_2 % 4294967296) / 2939745 / 4;
  uint32_t const Y   = 100 * C + Z;

  // Month and day.
  uint32_t const N_3 = 2141 * N_Y + 197913;
  uint32_t const M   = N_3 / 65536;
  uint32_t const D   = N_3 % 65536 / 2141;

  // Map. (Notice the year correction, including type change.)
  uint32_t const J   = N_Y >= 306;
  int32_t  const Y_G = (Y - L) + J;
  uint32_t const M_G = J ? M - 12 : M;
  uint32_t const D_G = D + 1;

  return { Y_G, M_G, D_G };
}
```

FIGURE 12 Function that finds the proleptic Gregorian date that is N_U days from 1 January 1970. It was derived to give correct results for, at least

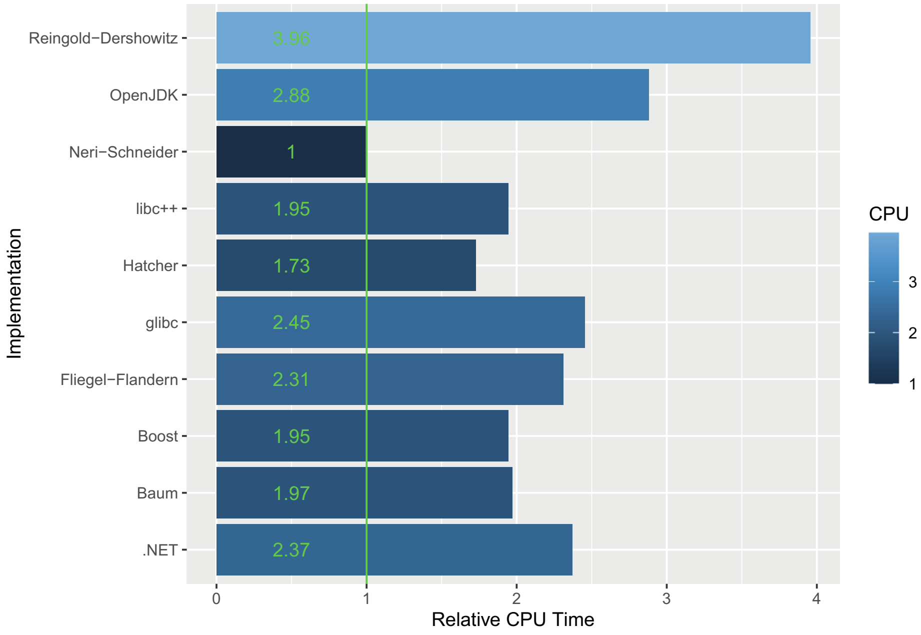
```
int32_t to_rata_die(int32_t Y_G, uint32_t M_G, uint32_t D_G) {

  // Shift and correction constants.
  uint32_t const s = 82;
  uint32_t const K = 719468 + 146097 * s;
  uint32_t const L = 400 * s;

  // Map. (Notice the year correction, including type change.)
  uint32_t const J = M_G <= 2;
  uint32_t const Y = (uint32_t(Y_G) + L) - J;
  uint32_t const M = J ? M_G + 12 : M_G;
  uint32_t const D = D_G - 1;
  uint32_t const C = Y / 100;

  // Rata die.
  uint32_t const y_star = 1461 * Y / 4 - C + C / 4;
  uint32_t const m_star = (979 * M - 2919) / 32;
  uint32_t const N      = y_star + m_star + D;

  // Rata die shift.
  uint32_t const N_U = N - K;

  return N_U;
}
```

FIGURE 13 Function that finds the number of days from 1 January 1970 and a given date of the proleptic Gregorian calendar. It was derived to give correct results for, at least, all dates from 1 January −32767 to 31 December 32767 (although the range of validity is much larger)



Implementation	Time (μs)
scanning only	21.76
Reingold-Dershowitz	531.25
OpenJDK	168.80
Neri-Schneider	83.70
libc++	184.17
Hatcher	157.29
glibc	481.41
Fliegel-Flandern	172.63
Boost	112.31
Baum	134.97
.NET	260.91

FIGURE 14 Relative and absolute timings of date calculations



Implementation	Time (μs)
scanning only	17.26
Reingold-Dershowitz	131.14
OpenJDK	100.10
Neri-Schneider	46.04
libc++	73.24
Hatcher	66.96
glibc	87.89
Fliegel-Flandern	83.71
Boost	73.24
Baum	73.94
.NET	85.45

FIGURE 15 Relative and absolute timings of *rata die* evaluations.

that $q = n/d$ and $r = n\%d$. However, if *in addition* we prove that $0 < r < |d|$, then the uniqueness allows this conclusion. Therefore, in many of the following proofs, we will first decompose a given n as $n = d \cdot q + r$ and subsequently prove that $0 < r < |d|$ to conclude that $q = n/d$ and $r = n\%d$.

Lemma 1 (Residual lemma). *Let $f(n) = (a \cdot n + b)/d$, with $d > 0$, and $g : \mathbb{Z} \to \mathbb{Z}$. For any $q \in \mathbb{Z}$ such that $f(g(q) - 1) < f(g(q))$, we have*:

$$n \in \mathbb{Z} \quad \text{and} \quad f(n) = f(g(q)) \quad \Rightarrow \quad f^{\circ}(n) = n - g(q).$$

TABLE 4 Relative CPU times for several platforms. The baseline (time = 1) is the code of Figure 13

Platform	.NET	Baum	Boost	Fliegel -Flandern	glibc	Hatcher	libc++	OpenJDK	Reingold-Dershowitz
clang_11.0.0-linux-intel_i7_10510U	2.29	1.65	1.89	2.51	2.55	2.58	1.79	3.16	3.56
clang_11.0.0-linux-ryzen_7_1800X	1.74	1.59	1.78	2.25	2.27	2.39	1.77	2.62	3.43
clang_12.0.0-windows-ryzen_7_1800X	1.79	1.57	1.62	2.39	2.16	1.88	1.88	2.56	4.61
clang_14.0.6-linux-intel_i7_10510U	1.90	1.49	1.43	2.27	2.26	1.75	1.60	2.69	4.09
clang_14.0.6-linux-ryzen_7_1800	1.83	1.62	1.62	2.41	2.32	1.87	1.83	2.81	4.79
gcc_10.2.0-linux-intel_i7_10510U	2.40	1.48	1.76	2.34	2.85	2.28	1.67	3.09	3.39
gcc_10.2.0-linux-ryzen_7_1800X	1.97	1.55	1.71	2.13	2.82	2.15	1.65	2.77	3.22
gcc_12.1.0-linux-intel_i7_10510U	2.36	1.57	1.50	2.43	3.02	2.04	1.66	3.35	4.19
gcc_12.1.0-linux-ryzen_7_1800X	1.87	1.47	1.52	2.19	2.77	1.99	1.60	2.81	3.90
msvc_19.29-windows-ryzen_7_1800X	2.65	2.20	2.17	2.40	2.65	1.93	2.07	3.05	4.52
msvc_19.29-windows-intel_i7_8750H	2.37	1.97	1.95	2.31	2.45	1.73	1.95	2.88	3.96

Proof. For any $x \in \mathbb{Z}$ we have $x\%d = x - d \cdot (x/d)$, which applied to arbitrary x_1 and x_2 yields:

$$x_1\%d - x_2\%d = \left[x_1 - d \cdot (x_1/d)\right] - \left[x_2 - d \cdot (x_2/d)\right] = (x_1 - x_2) - d \cdot (x_1/d - x_2/d).$$

In particular, for $x_1 = a \cdot n + b$ and $x_2 = a \cdot g(q) + b$ we obtain:

$$\begin{aligned}
(a \cdot n + b)\%d - (a \cdot g(q) + b)\%d &= a \cdot (n - g(q)) - d \cdot \left[(a \cdot n + b)/d - (a \cdot g(q) + b)/d\right] \\
&= a \cdot (n - g(q)) - d \cdot \left[f(n) - f(g(q))\right] && \text{(by definition of } f\text{)} \\
&= a \cdot (n - g(q)) && \text{(from } f(n) = f(g(q))\text{.)} \\
\therefore \qquad (a \cdot n + b)\%d &= a \cdot (n - g(q)) + (a \cdot g(q) + b)\%d
\end{aligned}$$

Provided we show that $0 \leq (a \cdot g(q) + b)\%d < a$ it will follow that $(a \cdot n + b)\%d/a = n - g(q)$, that is, $f^\circ(n) = n - g(q)$.

Trivially, $0 \leq (a \cdot g(q) + b)\%d$ as it is a result of %. Now, suppose by contradiction that $(a \cdot g(q) + b)\%d \geq a$. Then,

$$\begin{aligned}
a \cdot (g(q) - 1) + b &= (a \cdot g(q) + b) - a \\
&= d \cdot \left[(a \cdot g(q) + b)/d\right] + (a \cdot g(q) + b)\%d - a && \text{(by Euclidean division } (a \cdot g(q) + b)/d\text{)} \\
&= d \cdot f(g(q)) + (a \cdot g(q) + b)\%d - a && \text{(by definition of } f\text{)} \\
&\geq d \cdot f(g(q)) && \text{(from the assumption } (a \cdot g(q) + b)\%d \geq a\text{.)} \\
\therefore \quad (a \cdot (g(q) - 1) + b)/d &\geq f(g(q)) && \text{(by dividing both sides by } d\text{.)} \\
\therefore \quad f(g(q)) &\geq f(g(q) - 1) && \text{(by definition of } f\text{.)}
\end{aligned}$$

The above contradicts the assumption on q. ∎

The following is a more complete form of Theorem 1.

Theorem 6 (EAF Theorem complete form). *Let $f(n) = (a \cdot n + b)/d$ with $d \geq a > 0$. Then, for any $n \in \mathbb{Z}$ and $q \in \mathbb{Z}$ we have*:

1. *$f(f^*(q)) = q$ and $f(f^*(q) - 1) = q - 1$;*
2. *$f(n) = q$ if, and only if, $n \in [f^*(q), f^*(q + 1)[$;*
3. *$n \in [f^*(f(n)), f^*(f(n) + 1)[$ and $f^\circ(n) = n - f^*(f(n))$.*

Proof. (1) Since $0 \leq (a^* \cdot q + b^*)\%a \leq a - 1$ and $a \leq d$, we have:

$$0 \leq a - 1 - (a^* \cdot q + b^*)\%a \leq d - 1 - (a^* \cdot q + b^*)\%a < d. \tag{21}$$

We also have:

$$\begin{aligned}
a \cdot f^*(q) + b &= a \cdot \left[(a^* \cdot q + b^*)/d^*)\right] + b && \text{(by definition of } f^*\text{)} \\
&= a \cdot \left[(a^* \cdot q + b^*)/a)\right] + b && \text{(from } d^* = a\text{)} \\
&= a^* \cdot q + b^* - (a^* \cdot q + b^*)\%a + b && \text{(from } a \cdot (x/a) = x - x\%a,\ \forall x \in \mathbb{Z}\text{)} \\
&= d \cdot q + \left[a - 1 - (a^* \cdot q + b^*)\%a\right] && \text{(from } a^* = d \text{ and } b^* = a - b - 1.\text{)} \\
\therefore \quad (a \cdot f^*(q) + b)/d &= q && \text{(from Eq. (21))} \\
\therefore \quad f(f^*(q)) &= q && \text{(by definition of } f\text{)} \qquad (22)
\end{aligned}$$

This is the first conclusion of Item 1. Now, subtracting a from both sides of Equation (32) gives:

$$\begin{aligned}
a \cdot (f^*(q) - 1) + b &= d \cdot q - 1 - (a^* \cdot q + b^*)\%a \\
&= d \cdot (q - 1) + \left[d - 1 - (a^* \cdot q + b^*)\%a\right] && \text{(by subtracting and adding } d \text{ to the right side.)} \\
\therefore \quad (a \cdot (f^*(q) - 1) + b)/d &= q - 1 && \text{(from Eq. (21))} \\
\therefore \quad f(f^*(q) - 1) &= q - 1 && \text{(by definition of } f\text{)}
\end{aligned}$$

This concludes the proof of Item 1.

(2) Since $d > 0$ and $a > 0, f(n) = (a \cdot n + b)/d$ is non-decreasing and, thus, the following holds:

$$n \leq f^*(q) - 1 \quad \Rightarrow \quad f(n) \leq f(f^*(q) - 1) = q - 1 \qquad \text{(the equality comes from Item 1)} \tag{23}$$

$$f^*(q) \leq n \quad \Rightarrow \quad q = f(f^*(q)) \leq f(n) \qquad \text{(ditto)} \tag{24}$$

$$n \leq f^*(q + 1) - 1 \quad \Rightarrow \quad f(n) \leq f(f^*(q + 1) - 1) = q \qquad \text{(the equality comes from Item 1 applied to } q + 1\text{)} \tag{25}$$

$$f^*(q + 1) \leq n \quad \Rightarrow \quad q + 1 f(f^*(q + 1)) \leq f(n) \qquad \text{(ditto.)} \tag{26}$$

Equations (23) and (26) show that if $n \notin [f^*(q), f^*(q + 1)[$, then $f(n) \neq q$ whereas Eqs. (24) and (25) show that if $n \in [f^*(q), f^*(q + 1)[$, then $f(n) = q$. This concludes Item 2.

(3) Since $d \geq a > 0$, we have $a^* \geq d^* > 0$ and $f^*(q) = (a^* \cdot q + b^*)/d^*$ is strictly increasing. Then for q large enough, we have $n < f^*(q + 1)$ and we pick the minimum such q. From the minimality of q we obtain $f^*(q) \leq n$, that is, $n \in [f^*(q), f^*(q + 1)[$. Hence, Item 2 yields $f(n) = q$ and therefore, $n \in [f^*(f(n)), f^*(f(n) + 1)[$.

We will show that f^* fulfills the hypotheses on g of Lemma 1 and then conclude that $f^\circ(n) = n - f^*(q) = n - f^*(f(n))$. Item 1 gives $f(f^*(q) - 1) = q - 1 < q = f(f^*(q))$ and from $f(n) = q$ and Item 1 again, we obtain $f(n) = f(f^*(q))$. ■

14 | PROOFS OF RESULTS OF SECTION 7

14.1 | Fast evaluation of Euclidean affine functions

For ease of reference, we restate Theorems 2 and 3 before giving their proofs.

Theorem 2 (Fast round-up EAF evaluation). *Let $k \in \mathbb{Z}^+$ and $f(n) = (a \cdot n + b)/d$ with $d > 0$. Set $a' = 2^k \cdot a/d + 1$, $b' = -\min\left\{a' \cdot n - 2^k \cdot f(n)\ ;\ n \in [0, d[\right\}$ and $\varepsilon = d - 2^k \cdot a\%d$. For $n \in [0, d[$ define*:

$$Q(n) = \min\left\{q \in \mathbb{Z}^+\ ;\ \varepsilon \cdot q \geq 2^k \cdot (1 + f(n)) - (a' \cdot n + b')\right\} \quad \text{and} \quad P(n) = d \cdot Q(n) + n.$$

Let $U = \min\{P(n)\ ;\ n \in [0, d[\ \}$. *Then,*

$$(a \cdot n + b)/d = \left(a' \cdot n + b'\right)/2^k, \quad \forall n \in [0, U[.$$

Proof. Let $n \in \mathbb{Z}$ and note that $\varepsilon > 0$. Hence, if q is large enough, then $\varepsilon \cdot q$ becomes greater than $2^k \cdot (1 + f(n)) - (a' \cdot n + b')$. Therefore, $Q(n)$ is well-defined and so are $P(n)$ and U. Furthermore, $Q(n) \geq 0$ and, since $d > 0$, we also have $P(n) \geq 0$ and, consequently, $U \geq 0$. We do not exclude the possibility that $U = 0$, in which case this theorem's conclusion is vacuously true. The sequel assumes that $U > 0$ and $n \in [0, U[$. We have:

$$\begin{aligned}
a' \cdot n + b' &= a' \cdot \{d \cdot (n/d) + n\%d\} + b' && \text{(by division of } n\text{by } d\text{)}\\
&= \left\{a' \cdot d - 2^k \cdot a\right\} \cdot (n/d) + 2^k \cdot a \cdot (n/d) + a' \cdot (n\%d) + b' && \text{(by subtracting and adding } 2^k \cdot a \cdot (n/d)\text{)}\\
&= \left\{\left[1 + 2^k \cdot a/d\right] \cdot d - 2^k \cdot a\right\} \cdot (n/d) + 2^k \cdot a \cdot (n/d) + a' \cdot (n\%d) + b' && \text{(by definition of } a'\text{)}\\
&= \left\{d - \left[2^k \cdot a - (2^k \cdot a/d) \cdot d\right]\right\} \cdot (n/d) + 2^k \cdot a \cdot (n/d) + a' \cdot (n\%d) + b' &&\\
&= \left\{d - 2^k \cdot a\%d\right\} \cdot (n/d) + 2^k \cdot a \cdot (n/d) + a' \cdot (n\%d) + b' && \text{(by division of } 2^k \cdot a \text{ by } d\text{))}\\
&= \varepsilon \cdot (n/d) + 2^k \cdot a \cdot (n/d) + a' \cdot (n\%d) + b' && \text{(by definition of } \varepsilon.\text{)}\\
&= \varepsilon \cdot (n/d) + 2^k \cdot \{a \cdot (n/d) + f(n\%d)\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' && \text{(by adding and subtracting } 2^k \cdot f(n\%d)\text{)}\\
&= \varepsilon \cdot (n/d) + 2^k \cdot \left\{a \cdot (n/d) + \left[a \cdot (n\%d) + b\right]/d\right\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' && \text{(by definition of } f(n\%d)\text{)}\\
&= \varepsilon \cdot (n/d) + 2^k \cdot \left\{\left[a \cdot d \cdot (n/d) + a \cdot (n\%d) + b\right]/d\right\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' && \text{(since } a \cdot d \cdot (n/d) \text{ is multiple of } d\text{)}\\
&= \varepsilon \cdot (n/d) + 2^k \cdot \left\{\left[a \cdot (d \cdot (n/d) + (n\%d)) + b\right]/d\right\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' &&\\
&= \varepsilon \cdot (n/d) + 2^k \cdot \{(a \cdot n + b)/d\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' && \text{(by division of } n \text{ by } d\text{)}\\
&= 2^k \cdot \{(a \cdot n + b)/d\} + \left[\varepsilon \cdot (n/d) + a' \cdot (n\%d) + b' - 2^k \cdot f(n\%d)\right]. && (27)
\end{aligned}$$

Set $r = \varepsilon \cdot (n/d) + a' \cdot (n\%d) + b' - 2^k \cdot f(n\%d)$, the term inside the square brackets of the last equation, so that $a' \cdot n + b' = 2^k \cdot \{(a \cdot n + b)/d\} + r$. We shall show that $0 \leq r < 2^k$ and it will follow that $(a' \cdot n + b')/2^k = (a \cdot n + b)/d$, which concludes the proof.

Since $n\%d \in [0, d[$, the definition of b' yields $-b \leq a' \cdot (n\%d) - 2^k \cdot f(n\%d)$, or equivalently, $0 \leq a' \cdot (n\%d) + b' - 2^k \cdot f(n\%d)$. This and $\varepsilon \cdot (n/d) \geq 0$ give $r \geq 0$. Now, by definition, $U \leq P(n\%d)$ and thus, $n < P(n\%d)$. From $n = d \cdot (n/d) + n\%d$ and $P(n\%d) = d \cdot Q(n\%d) + n\%d$ we obtain $d \cdot (n/d) + n\%d < d \cdot Q(n\%d) + n\%d$, so that $n/d < Q(n\%d)$. From the minimality of $Q(n\%d)$ and the fact that $0 \leq n/d < Q(n\%d)$, we obtain:

$$\begin{aligned}
&\varepsilon \cdot (n/d) < 2^k \cdot (1 + f(n\%d)) - (a' \cdot (n\%d) + b')\\
\therefore\quad &\varepsilon \cdot (n/d) + a' \cdot (n\%d) + b' - 2^k \cdot f(n\%d) < 2^k.
\end{aligned}$$

And it follows that $r < 2^k$. ■

Theorem 3 (Fast round-down EAF evaluation). *Let* $k \in \mathbb{Z}^+$ *and* $f(n) = (a \cdot n + b)/d$ *with* $d > 0$ *and* $2^k \cdot a\%d > 0$. *Set* $a' = 2^k \cdot a/d$ *and* $b' = \min\left\{2^k - 1 - a' \cdot n + 2^k \cdot f(n)\ ;\ n \in [0, d[\right\}$ *and* $\varepsilon = 2^k \cdot a\%d$. *For* $n \in [0, d[$ *define*:

$$Q(n) = \min\left\{q \in \mathbb{Z}^+\ ;\ \varepsilon \cdot q > (a' \cdot n + b') - 2^k \cdot f(n)\right\} \quad \text{and} \quad P(n) = d \cdot Q(n) + n.$$

Let $U = \min\{P(n)\ ;\ n \in [0, d[\ \}$. *Then,*

$$(a \cdot n + b)/d = (a' \cdot n + b')/2^k, \quad \forall n \in [0, U[.$$

Proof. Let $n \in \mathbb{Z}$. Since $\varepsilon > 0$, if q is large enough, then $\varepsilon \cdot q$ becomes greater than $(a' \cdot n + b') - 2^k \cdot f(n)$. Therefore, $Q(n)$ is well-defined and so are $P(n)$ and U. Furthermore, $Q(n) \geq 0$ and, since $d > 0$, we also have $P(n) \geq 0$ and, consequently, $U \geq 0$. We do not exclude the possibility that $U = 0$, in which case this theorem's conclusion is vacuously true. The sequel assumes that $U > 0$ and $n \in [0, U[$. We have:

$$
\begin{aligned}
a' \cdot n + b' &= a' \cdot \left[d \cdot (n/d) + n\%d\right] + b' && \text{(by division of } n \text{ by } d\text{)}\\
&= \left[a' \cdot d - 2^k \cdot a\right] \cdot (n/d) + 2^k \cdot a \cdot (n/d) + a' \cdot (n\%d) + b' && \text{(by subtracting and adding } 2^k \cdot a \cdot (n/d)\text{)}\\
&= \left[(2^k \cdot a/d) \cdot d - 2^k \cdot a\right] \cdot (n/d) + 2^k \cdot a \cdot (n/d) + a' \cdot (n\%d) + b' && \text{(by definition of } a'\text{)}\\
&= \left[-2^k \cdot a\%d\right] \cdot (n/d) + 2^k \cdot a \cdot (n/d) + a' \cdot (n\%d) + b' && \text{(by division of } 2^k \cdot a \text{ by } d\text{))}\\
&= -\varepsilon \cdot (n/d) + 2^k \cdot a \cdot (n/d) + a' \cdot (n\%d) + b' && \text{(by definition of } \varepsilon.\text{)}\\
&= -\varepsilon \cdot (n/d) + 2^k \cdot \{a \cdot (n/d) + f(n\%d)\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' && \text{(by adding and subtracting } 2^k \cdot f(n\%d)\text{)}\\
&= -\varepsilon \cdot (n/d) + 2^k \cdot \left\{a \cdot (n/d) + \left[a \cdot (n\%d) + b\right]/d\right\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' && \text{(by definition of } f(n\%d)\text{)}\\
&= -\varepsilon \cdot (n/d) + 2^k \cdot \left\{\left[a \cdot d \cdot (n/d) + a \cdot (n\%d) + b\right]/d\right\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' && \text{(since } a \cdot d \cdot (n/d) \text{ is multiple of } d\text{)}\\
&= -\varepsilon \cdot (n/d) + 2^k \cdot \left\{\left[a \cdot (d \cdot (n/d) + (n\%d)) + b\right]/d\right\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' &&\\
&= -\varepsilon \cdot (n/d) + 2^k \cdot \{(a \cdot n + b)/d\} - 2^k \cdot f(n\%d) + a' \cdot (n\%d) + b' && \text{(by division of } n \text{ by } d\text{)}\\
&= 2^k \cdot \{(a \cdot n + b)/d\} + \left[-\varepsilon \cdot (n/d) + a' \cdot (n\%d) + b' - 2^k \cdot f(n\%d)\right].
\end{aligned}
$$

Set $r = -\varepsilon \cdot (n/d) + a' \cdot (n\%d) + b' - 2^k \cdot f(n\%d)$, the term inside the square brackets of the last equation, so that $a' \cdot n + b' = 2^k \cdot \{(a \cdot n + b)/d\} + r$. We shall show that $0 \le r < 2^k$ and it will follow that $(a' \cdot n + b')/2^k = (a \cdot n + b)/d$, which concludes the proof.

Since $n\%d \in [0, d[$, the definition of b' yields $b' \le 2^k - 1 - a' \cdot (n\%d) + 2^k \cdot f(n\%d)$, or equivalently, $a' \cdot (n\%d) + b' - 2^k \cdot f(n\%d) \le 2^k - 1$. This and $\varepsilon \cdot (n/d) \ge 0$ give $r \le 2^k - 1$. Now, by definition, $U \le P(n\%d)$ and thus, $n < P(n\%d)$. From $n = d \cdot (n/d) + n\%d$ and $P(n\%d) = d \cdot Q(n\%d) + n\%d$ we get $d \cdot (n/d) + n\%d < d \cdot Q(n\%d) + n\%d$, so that $n/d < Q(n\%d)$. From the minimality of $Q(n\%d)$ and the fact that $0 \le n/d < Q(n\%d)$, we obtain:

$$
\begin{aligned}
&\varepsilon \cdot (n/d) \le (a' \cdot (n\%d) + b') - 2^k \cdot f(n\%d)\\
\therefore\quad &0 \le -\varepsilon \cdot (n/d) + a' \cdot (n\%d) + b' - 2^k \cdot f(n\%d).
\end{aligned}
$$

And it follows that $r \ge 0$. ■

Recall that by applying Theorems 2 and 3 in Examples 8 and 9 we obtained two fast alternatives for the same EAF valid on $[0, 12[$ and $[0, 34[$, respectively. Hence, the larger interval was obtained with Theorem 3 but sometimes it is Theorem 3 that yields the larger interval. For EAFs known prior to compilation, we can find both alternatives and select the one that is valid in the larger range. A simple criterion for guessing in advance the theorem that will give the larger interval is based on the following heuristics. To maximise U we seek to maximise $Q(N)$, which is the minimum value of q for which $\varepsilon \cdot q$ reaches a certain threshold. The smaller ε is, the larger q must be for this to happen. Hence, it is likely that a larger interval will be obtained for the alternative with the smallest ε amongst its two possible values, namely, $\varepsilon_1 = d - 2^k \cdot a\%d$ and $\varepsilon_2 = 2^k \cdot a\%d$. (If d is odd, then $\varepsilon_1 \ne \varepsilon_2$.) Another interpretation of this criterion is that it sets a' to the best approximation of $2^k \cdot a \cdot d^{-1}$ given by $\lceil 2^k \cdot a \cdot d^{-1} \rceil$ or $\lfloor 2^k \cdot a \cdot d^{-1} \rfloor$. Indeed, suppose that $2^k \cdot a \cdot d^{-1}$ is not integer, so that $\lceil 2^k \cdot a \cdot d^{-1} \rceil = 2^k \cdot a/d + 1$ and $\lfloor 2^k \cdot a \cdot d^{-1} \rfloor = 2^k \cdot a/d$. Then:

$$
\begin{aligned}
\varepsilon_1 &= d - 2^k \cdot a\%d, & \varepsilon_2 &= 2^k \cdot a\%d,\\
&= d - \left[2^k \cdot a - (2^k \cdot a/d) \cdot d\right], & &= 2^k \cdot a - (2^k \cdot a/d) \cdot d,\\
&= (1 + 2^k \cdot a/d) \cdot d - 2^k \cdot a, & &\\
&= \left\lceil 2^k \cdot a \cdot d^{-1} \right\rceil \cdot d - 2^k \cdot a, & &= 2^k \cdot a - \left\lfloor 2^k \cdot a \cdot d^{-1} \right\rfloor \cdot d.
\end{aligned}
$$

It follows that $\varepsilon_1 < \varepsilon_2$ if, and only if, $\lceil 2^k \cdot a \cdot d^{-1} \rceil - 2^k \cdot a \cdot d^{-1} < 2^k \cdot a \cdot d^{-1} - \lfloor 2^k \cdot a \cdot d^{-1} \rfloor$.

14.2 | Fast division – the special case $a = 1, b = 0$

We again turn our attention to division n/d with $d > 0$, that is, the particular case of the EAF $f(n) = (a \cdot n + b)/d$ where $a = 1$, $b = 0$ and $d > 0$ and d is not a power of two.

For $a = 1$, Theorems 2 and 3 set a' to $\lceil 2^k \cdot d^{-1} \rceil$ and $\lfloor 2^k \cdot d^{-1} \rfloor$, respectively. The former is the choice made by Alverson[4] Cavagnino and Werbrouck,[5] and Granlund and Montgomery,[6] while the latter is used by Magenheimer.[7] Finally, Robison[8] and an appendix to Cavagnino and Werbrouck[5] consider both, and in this particular case, rigourously justify the heuristics we have suggested for choosing between the two approaches.

This section follows a more direct path but most of its results can be obtained from Theorems 2 and 3 for $a = 1$ and $b = 0$. For instance, for $a' > 0$, let $b' = -\min\{a' \cdot n - 2^k \cdot f(n)\ ;\ n \in [0, d[\ \}$ as in Theorem 2. Since $f(n) = n/d = 0$, for $n \in [0, d[$, we have $b' = -\min\{a' \cdot n\ ;\ n \in [0, d[\ \} = 0$. Hence, in contrast to the general case, there is no need for an $O(d)$ search to obtain b'. Similarly, $b' = \min\{2^k - 1 - a' \cdot n + 2^k \cdot f(n)\ ;\ n \in [0, d[\ \}$, as defined by Theorem 3, can be proven to be $a' + 2^k\%d - 1$, the same value found in Magenheimer et al.[7] Theorem 3 assumes that $2^k\%d \geq 1$ and, in the definition of b', had we subtracted $2^k\%d$ instead of 1, the proof would still work but b' would have a smaller value, namely, $b' = a'$. In this case, the final reduction would be $n/d = a' \cdot (n+1)/2^k$ as found in Cavagnino and Werbrouck[5] and in Robison.[8] In addition, by making b' smaller, $Q(n)$, $P(n)$ and U can also decrease. Hence, Theorem 3 obtains a range of validity that is no smaller than the one obtained in in Cavagnino and Werbrouck[5] and in Robison.[8]

The remainder of this section focuses on the round-up approach, but the round-down alternative could be similarly considered. Our reasons for this are that the round-up approach is mostly used by compilers and that the appearance of the term $n + 1$ can lead to an overflow if it is not dealt with appropriately.

Lemma 2. *Let $d, k \in \mathbb{Z}^+$ with $d > 0$. Set $a' = 2^k/d + 1$ and $\varepsilon = d - 2^k\%d$. Then, $a' \cdot d = 2^k + \varepsilon$ and for any $n \in \mathbb{Z}$ we have*:

$$a' \cdot n/2^k = n/d \quad \text{if, and only if,} \quad 0 \leq \varepsilon \cdot (n/d) + a' \cdot (n\%d) < 2^k. \tag{28}$$

Proof. Taking $a = 1$ and $b = 0$ in Theorem 2 gives the same a' and ε as here. Now,

$$\begin{aligned}
a' \cdot d &= (2^k/d + 1) \cdot d && \text{(by definition of } a'\text{)}\\
&= (2^k/d) \cdot d + d\\
&= 2^k - 2^k/\%d + d && \text{(by division of } 2^k \text{ by } d\text{)}\\
&= 2^k + \varepsilon && \text{(by definition of } \varepsilon.\text{)}
\end{aligned}$$

Similarly, we have $b' = 0$ and $f(n\%d) = n\%d/d = 0$. From Equation (27) with $f(n\%d) = n\%d/d = 0$ we obtain:

$$a' \cdot n - 2^k \cdot (n/d) = \varepsilon \cdot (n/d) + a' \cdot (n\%d).$$

The result follows from Euclidean division by 2^k. ■

Since $a', \varepsilon \in \mathbb{Z}^+$, we have $0 \leq \varepsilon \cdot (n/d) + a' \cdot (n\%d)$ for all $n \in \mathbb{Z}^+$. There is an illuminating geometric interpretation for the second part of the double inequality in Equation (28) that follows from mapping each $n \in \mathbb{Z}^+$ to $P_n = (n/d, n\%d)$ in the *xy* plane. Figure 16 shows, for $d = 5$, some points of particular interest, namely, P_0, P_1, P_3, P_4, P_5, and P_{64}, respectively pictured by their circled values: ⓪, ①, ③, ④, ⑤, and (64). Inequality $\varepsilon \cdot (n/d) + a' \cdot (n\%d) < 2^k$ states that P_n is below the line $\varepsilon \cdot x + a' \cdot y = 2^k$, which is represented by the dotted line in Figure 16, for $k = 8$, $\varepsilon = 4$ and $a' = 52$. Hence, Equation (28) means that a point below the line corresponds to $n \in \mathbb{Z}^+$ such that $n/d = a' \cdot n/2^k$ and a point above or on the line is related to n such that $n/d \neq a' \cdot n/2^k$.

If the slope of the dotted line is not too steep (more precisely, $-\varepsilon \cdot a'^{-1} \geq -1$), then the graph makes it obvious that the smallest U for which P_U is above or on the dotted line must lie on the line $y = d - 1 = 4$. In our case $U = 64$. For Theorem 2, with $a = 1$ and $b = 0$, this means that $U = P(d - 1)$. In particular, we are not interested in either $P(n)$ or $Q(n)$

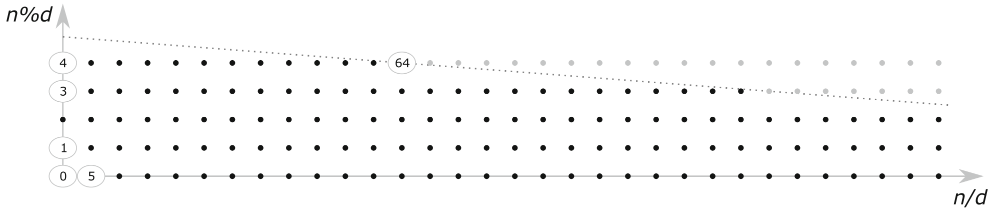


FIGURE 16 The geometry of replacing n/d with $a' \cdot n/2^k$, where $d = 5$ and $k = 8$, $a' = 2^k/d + 1 = 52$

when $n \neq d - 1$. This and $b' = 0$ turn finding a fast EAF and its interval of applicability $[0, U[$ into an $O(1)$ calculation rather than an $O(d)$ search as in the general case. These geometric ideas are present, although algebraically disguised, in the proof of Theorem 4.

The result of Theorem 4 also appears in Cavagnino and Werbrouck.[5] In addition to providing the aforementioned geometric insights and an arguably simpler proof, we favour faster algorithms over applicability range. In other words, we reduce division to multiplication and bitwise shift and find the largest U for which this optimisation yields correct results for all dividends in $[0, U[$. (In Cavagnino and Werbrouck,[5] U is called *critical value* and is denoted by N_{cr}.)

Theorem 4 (Fast division). *Let $d, k \in \mathbb{Z}^+$ with $d > 0$. Set $a' = 2^k/d + 1$, $\varepsilon = d - 2^k\%d$ and $U = \lceil a' \cdot \varepsilon^{-1} \rceil \cdot d - 1$. If $\varepsilon \leq a'$, then*:

$$n/d = a' \cdot n/2^k, \quad \forall n \in [0, U[.$$

Proof. Let $n \in [0, U[$. From Lemma 2, it suffices to show that:

$$0 \leq \varepsilon \cdot (n/d) + a' \cdot (n\%d) < 2^k \tag{29}$$

Since ε, a' and n are non-negative, the first inequality above (positiveness) is trivially true. We shall prove the second inequality in two steps: $n \leq U - d$ and $U - d + 1 \leq n$ but, before this, note that U can be written as $U = \left(\lceil a' \cdot \varepsilon^{-1} \rceil - 1\right) \cdot d + (d - 1)$, which gives:

$$U/d = \lceil a' \cdot \varepsilon^{-1} \rceil - 1 \qquad \text{and} \qquad U\%d = d - 1. \tag{30}$$

Assume first that $n \leq U - d$, so that $n/d \leq U/d - 1$. We have:

$$\begin{aligned}
\varepsilon \cdot (n/d) + a' \cdot (n\%d) &\leq \varepsilon \cdot (U/d - 1) + a' \cdot (d - 1) && \text{(from } n/d \leq U/d - 1 \text{ and } n\%d \leq d - 1\text{)} \\
&= \varepsilon \cdot \left(\lceil a' \cdot \varepsilon^{-1} \rceil - 2\right) + a' \cdot (d - 1) && \text{(from Equation (30))} \\
&= \varepsilon \cdot \lceil a' \cdot \varepsilon^{-1} \rceil - 2 \cdot \varepsilon + a' \cdot d - a' \\
&< a' + \varepsilon - 2 \cdot \varepsilon + a' \cdot d - a' && \text{(from } \lceil a' \cdot \varepsilon^{-1} \rceil < a' \cdot \varepsilon^{-1} + 1\text{)} \\
&= a' \cdot d - \varepsilon \\
&= 2^k && \text{(from } a' \cdot d = 2^k + \varepsilon \text{ as seen in Lemma 2.)} \\
\therefore \quad \varepsilon \cdot (n/d) + a' \cdot (n\%d) &< 2^k.
\end{aligned}$$

Now assume that $U - d + 1 \leq n < U$ and set $r = n - (U - d + 1)$, so that $0 \leq r < d - 1$ and $n = (U - d + 1) + r$. Therefore:

$$\begin{aligned}
n &= U - (d - 1) + r \\
&= U - U\%d + r && \text{(from Equation (30))} \\
&= (U/d) \cdot d + r && \text{(by division of } U \text{ by } d.\text{)}
\end{aligned}$$

Therefore $n/d = U/d$ and $n\%d = r$ and it follows that:

$$\begin{aligned}
\varepsilon \cdot (n/d) + a' \cdot (n\%d) &\leq \varepsilon \cdot (U/d) + a' \cdot (d - 2) && \text{(since } r \leq d - 2\text{)} \\
&= \varepsilon \cdot (U/d - 1) + a' \cdot (d - 1) + \varepsilon - a' && \text{(by subtracting and adding } \varepsilon\text{)} \\
&\leq \varepsilon \cdot (U/d - 1) + a' \cdot (d - 1) && \text{(from assumption: } \varepsilon \leq a'\text{)} \\
&< 2^k && \text{(similar to the case } n \leq U - d.\text{)}
\end{aligned}$$

■

Remark 2. If d is not a power of two and $2^k \geq d \cdot (d - 2)$, then $\varepsilon \leq a'$. Indeed, under these hypotheses we have $2^k/d \geq d - 2$ and $2^k\%d \geq 1$. It follows that $d - 2^k\%d \leq d - 1 \leq 2^k/d + 1$, that is, $\varepsilon \leq a'$.

We have already applied Theorem 4 in the derivation of Algorithm 5 and the next example shows other divisions that frequently appear in time calculations and decimal expansions.

Example 14 (Time calculations).

$$
\begin{aligned}
n/3600 &= 1193047 \cdot n/2^{32}, && \forall n \in [0, 2257199[,\\
n/60 &= 71582789 \cdot n/2^{32}, && \forall n \in [0, 97612919[,\\
n/10 &= 429496730 \cdot n/2^{32}, && \forall n \in [0, 1073741829[.
\end{aligned}
$$

The first two lines can be used in conversions of seconds elapsed since midnight, a quantity in $[0, 86400[$, to hours, minutes and seconds. The third line can be used in conversions of non-negative integers up to 9 digits into their decimal representations.

14.3 | Fast evaluation of residual functions

For ease of reference, we restate Theorem 5 here and then provide its proof.

Theorem 5 (Alternative residual evaluation). *Let $f(n) = (a \cdot n + b)/d$ and $f'(n) = (a' \cdot n + b')/d'$, with $d \geq a > 0$, and assume that $f(n) = f'(n)$ for all $n \in [L, U[$. If $f^*(f(L)) = L$ and $f'(L-1) < f'(L)$, then*:

$$f^\circ(n) = f^{\circ\prime}(n) \quad \forall n \in [L, U[.$$

Proof. Let $n \in [L, U[$, so that $f(n) = f'(n)$ and set $q = f(n) = f'(n)$. Theorem 6-(3) gives $f^\circ(n) = n - f^*(q)$ and we will show that $f^{\circ\prime}(n) = n - f^*(q)$ to conclude the proof. To do that, we will use Lemma 1 with f' instead of f and $g = f^*$, that is, it requires us to show the following:

(1) $f'(n) = f'(f^*(q))$; and
(2) $f'(f^*(q) - 1) < f'(f^*(q))$.

We will first show that $f^*(q) \in [L, U[$. Since $d \geq a > 0$, both f and f^* are non-decreasing. Hence, from $L \leq n \leq U - 1$, we obtain $f^*(f(L)) \leq f^*(f(n)) \leq f^*(f(U-1))$. By assumption, $L = f^*(f(L))$ and Theorem 6-(3) (for $n = U - 1$) gives, in particular, $f^*(f(U-1)) \leq U - 1$. Therefore, $L \leq f^*(f(n)) \leq U - 1$, in other words, $f^*(q) \in [L, U[$.

Item 14.3 is obtained as follows:

$$
\begin{aligned}
f'(f^*(q)) &= f(f^*(q)) && \text{(from } f \equiv f' \text{ on } [L, U[\text{ and } f^*(q) \in [L, U[)\\
&= q && \text{(from Theorem 6-(1))} \qquad (31)
\end{aligned}
$$

Since $f^*(q) \in [L, U[$ either $f^*(q) = L$ or $f^*(q) - 1 \in [L, U[$. In the former case, the assumption $f'(L-1) < f'(L)$ reads $f'(f^*(q) - 1) < f'(f^*(q))$, which is Item 14.3. In the latter case we have:

$$
\begin{aligned}
f'(f^*(q) - 1) &= f(f^*(q) - 1) && \text{(from } f \equiv f' \text{ on } [L, U[\text{ and } f^*(q) - 1 \in [L, U[)\\
&= q - 1 && \text{(from Theorem 6-(1))}\\
&< q\\
&= f'(f^*(q)) && \text{(from Eq. (31).)}
\end{aligned}
$$

This concludes Item 14.3. ■

Example 15. Expanding on Example 14, Theorem 5 gives:

$$
\begin{aligned}
n\%3600 &= 1193047 \cdot n\%2^{32}/1193047, && \forall n \in [0, 2257199[,\\
n\%60 &= 71582789 \cdot n\%2^{32}/71582789, && \forall n \in [0, 97612919[,\\
n\%10 &= 429496730 \cdot n\%2^{32}/429496730, && \forall n \in [0, 1073741829[.
\end{aligned}
$$

14.4 | Quick remainder – the special case $a = 1, b = 0$

Again for this particular case, the EAF $f(n) = (a \cdot n + b)/d$ simplifies to $f(n) = n/d$ and its residual function simplifies to $f^{\circ}(n) = n\%d$. This section uses Theorem 4 and Theorem 5 to derive an efficient way to calculate remainders.

Formally, Theorem 4 states how to replace n/d with $a' \cdot n/2^k$, where $a' \approx 2^k \cdot d^{-1}$ and $k \in \mathbb{Z}^+$. Theorem 5 then gives the equality $n\%d = (a' \cdot n\%2^k)/a'$ and Theorem 4, again, suggests replacing division by a' with multiplication by an approximation of $2^k \cdot a'^{-1}$ and division by 2^k. It turns out that $d \approx 2^k \cdot a'^{-1}$ is the approximation we need and we obtain $n\%d = d \cdot (a' \cdot n\%2^k)/2^k$. This is the idea behind our next theorem.

Theorem 7. *Let $d, k \in \mathbb{Z}^+$ with $d > 0$. Set $a' = 2^k/d + 1$, $\varepsilon = d - 2^k\%d$ and $U' = \lceil 2^k \cdot \varepsilon^{-1} \rceil$. If $\varepsilon \leq a'$, then:*

$$n\%d = d \cdot (a' \cdot n\%2^k)/2^k, \quad \forall n \in [0, U'[. \tag{32}$$

Proof. Set $U = \lceil a' \cdot \varepsilon^{-1} \rceil \cdot d - 1$ as in Theorem 4. We have:

$$\begin{aligned}
& U \geq a' \cdot \varepsilon^{-1} \cdot d - 1 && (\text{since } \lceil a' \cdot \varepsilon^{-1} \rceil \geq a' \cdot \varepsilon^{-1}) \\
& \quad = (2^k + \varepsilon) \cdot \varepsilon^{-1} - 1 && (\text{since } a' \cdot d = 2^k + \varepsilon \text{ from Lemma 2}) \\
& \quad = 2^k \cdot \varepsilon^{-1}. && \\
\therefore \quad & U \geq \lceil 2^k \cdot \varepsilon^{-1} \rceil && (\text{since } U \text{ is integer and greater than or equal to } 2^k \cdot \varepsilon^{-1}) \\
\therefore \quad & U \geq U' && (\text{by definition of } U'.)
\end{aligned}$$

From the above and Theorem 4, we obtain:

$$n/d = a' \cdot n/2^k, \quad \forall n \in [0, U'[. \tag{33}$$

Hence, for $f(n) = n/d$ and $f'(n) = a' \cdot n/2^k$, Equation (33) states that $f \equiv f'$ on $[0, U'[$. Simple calculations give $f^*(f(0)) = 0$ and $f'(-1) < f'(0)$. Therefore, Theorem 5 (for $a = 1$, $b = 0$ and $L = 0$) yields:

$$n\%d = (a' \cdot n\%2^k)/a', \quad \forall n \in [0, U'[. \tag{34}$$

Let $n \in [0, U'[$ and set $m = \varepsilon \cdot (n/d) + a' \cdot (n\%d)$. Equation (33) and Lemma 2 give:

$$0 \leq \varepsilon \cdot (n/d) + a' \cdot (n\%d) < 2^k, \qquad \text{i.e.,} \qquad 0 \leq m < 2^k. \tag{35}$$

We have:

$$\begin{aligned}
& a' \cdot n = a' \cdot \left[d \cdot (n/d) + n\%d \right] && \text{by division of } n \text{ by } d) \\
& \quad = a' \cdot d \cdot (n/d) + a' \cdot (n\%d) && \\
& \quad = (2^k + \varepsilon) \cdot (n/d) + a' \cdot (n\%d) && (\text{since } a' \cdot d = 2^k + \varepsilon \text{ from Lemma 2}) \\
& \quad = 2^k \cdot (n/d) + \left[\varepsilon \cdot (n/d) + a' \cdot (n\%d) \right] && \\
& \quad = 2^k \cdot (n/d) + m && (\text{by definition of } m.) \\
\therefore \quad & a' \cdot n\%2^k = m && (\text{by division of } a' \cdot n \text{ by } 2^k \text{ and Equation (35).)} \\
\therefore \quad & n\%d = m/a' && (\text{from Equation (34).)}
\end{aligned} \tag{36}$$

We will show that $m/a' = d \cdot m/2^k$ and, thus, from the last two equations above, it will follow that $n\%d = d \cdot (a' \cdot n\%2^k)/2^k$, which concludes this theorem.

To show that $m/a' = d \cdot m/2^k$, we will apply Lemma 2 but “it is a perversity, not of the authors, but of nature”[###] that the symbols a' and d therein and here are in interchanged positions. We must therefore verify the two inequalities of Equation (28), not only with m in place of n but, annoyingly, a' in place of d and vice-versa. In addition, we must use a “different” ε defined by $a' - 2^k\%a'$. The good news is that ε is, actually, the same. We have:

[###] We borrowed this phrase from Paul Richard Halmos.

$$\begin{aligned} 2^k &= d \cdot a' - \varepsilon && \text{(since } a' \cdot d = 2^k + \varepsilon \text{ from Lemma 2)} \\ &= (d-1) \cdot a' + (a' - \varepsilon) && \text{(by subtracting and adding } a'\text{).} \end{aligned} \tag{37}$$

Now, since $\varepsilon = d - 2^k\%d$ we have $\varepsilon > 0$ and, by assumption, $\varepsilon \leq a'$. Hence, $0 \leq a' - \varepsilon < a'$. From this, Equation (37) and division by a' we obtain $2^k/a' = d - 1$ and $2^k\%a' = a' - \varepsilon$. In other words, $d = 2^k/a' + 1$ and $\varepsilon = a' - 2^k\%a'$. Note that these two expressions are the same set by Lemma 2, again, with a' and d interchanged. Therefore, to apply this lemma and conclude the proof, we need to show that:

$$0 \leq \varepsilon \cdot (m/a') + d \cdot (m\%a') < 2^k. \tag{38}$$

From $m = \varepsilon \cdot (n/d) + a' \cdot (n\%d)$ and Equation (36) we obtain $m/a' = n\%d$ and $m\%a' = \varepsilon \cdot (n/d)$. Therefore:

$$\begin{aligned} 0 \leq \varepsilon \cdot (m/a') + d \cdot (m\%a') &= \varepsilon \cdot (n\%d) + d \cdot \varepsilon \cdot (n/d) && \text{(from } m/a' = n\%d \text{ and } \varepsilon \cdot (n/d) = m\%a'\text{)} \\ &= \varepsilon \cdot \left[n\%d + d \cdot (n/d)\right] && \\ &= \varepsilon \cdot n && \text{(by division of } n \text{ by } d\text{)} \\ &< 2^k. && \text{(otherwise } n \geq 2^k \cdot \varepsilon^{-1} \text{ and then, } n \geq \left\lceil 2^k \cdot \varepsilon^{-1} \right\rceil \\ & && = U' \text{ which contradicts } n < U'\text{.)} \end{aligned}$$

Which proves Equation (38) as required. ■

Example 16. Revisiting Example 15 and using Theorem 7 gives:

$$\begin{aligned} n\%3600 &= 3600 \cdot (1193047 \cdot n\%2^{32})/2^{32}, && \forall n \in [0, 2255761[; \\ n\%60 &= 60 \cdot (71582789 \cdot n\%2^{32})/2^{32}, && \forall n \in [0, 97612894[; \\ n\%10 &= 10 \cdot (429496730 \cdot n\%2^{32})/2^{32}, && \forall n \in [0, 1073741824[. \end{aligned}$$

The expressions on the right side of the equals sign provide efficient ways of evaluating remainders. However, greater benefits are achieved when they are used in conjunction with the quotient expressions presented in Example 14.

The equality in Equation (32) also appears in Lemire et al.[9] As in other works, it focuses on obtaining the value $k \in \mathbb{Z}^+$ for which the equality holds on an interval of the form $[0, 2^w[$ or, in other words, $2^w \leq U'$ as the next result shows.

Corollary 1. *Let $d, l, w \in \mathbb{Z}^+$ with $0 < d < 2^w$ and $d - 2^{w+l}\%d \leq 2^l$. Set $a' = 2^{w+l}/d + 1$. Then,*

$$n\%d = d \cdot (a' \cdot n\%2^{w+l})/2^{w+l}, \quad \forall n \in [0, 2^w[.$$

Proof. Set $k = w + l$, $\varepsilon = d - 2^k\%d$ and $U' = \left\lceil 2^k \cdot \varepsilon^{-1} \right\rceil$. From Theorem 7, it is sufficient to show that $\varepsilon \leq a'$ and $2^k \leq U'$. Since $d < 2^w$ and $\varepsilon \leq 2^l$, we have $d \cdot \varepsilon < 2^{w+l} = 2^k$ and thus, $\varepsilon \leq 2^k/d < a'$. We also have $U' \geq 2^k \cdot \varepsilon^{-1} \geq 2^k \cdot 2^{-l} = 2^w$. ■

ACKNOWLEDGMENTS

We thank the editor, Dr. Daniel Lemire, and an anonymous referee for their detailed review and insightful suggestions, which helped us to improve the quality of our article. We are also grateful to Cristina Acosta and Becky Rawlings for their continuing support and helpful feedback.

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

AUTHOR CONTRIBUTIONS

Conceptualization, methodology, investigation and writing were performed by Cassio Neri and Lorenz Schneider. Coding was done by Cassio Neri. Funding for the project was acquired by Lorenz Schneider.

ORCID

Cassio Neri https://orcid.org/0000-0001-6940-188X
Lorenz Schneider https://orcid.org/0000-0001-5278-8184

REFERENCES

1. Richards EG. *Mapping Time: The Calendar and its History*. Oxford University Press; 1998.
2. Duncan DE. *The Calendar: The 5000-Year Struggle to Align the Clock and the Heavens*. 1st ed. Fourth Estate; 1998.
3. Duncan S. *Marking Time: The Epic Quest to Invent the Perfect Calendar*. 1st ed. John Wiley & Sons; 2000.
4. Alverson R. Integer division using reciprocals. *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*. 1991; pp. 186-190.
5. Cavagnino D, Werbrouck AE. Efficient algorithms for integer division by constants using multiplication. *Comput J*. 2007;51(4):470-480.
6. Granlund T, Montgomery PL. Division by invariant integers using multiplication. Proceedings of the ACM SIGPLAN. *Conference on Programming Language Design and Implementation: 61–72; 1994*. New York, NY, USA; 1994.
7. Magenheimer DJ, Peters L, Pettis KW, Zuras D. Integer multiplication and division on the HP precision architecture. *IEEE Trans Comput*. 1988;37(8):980-990.
8. Robison AD. N-bit unsigned division via n-bit multiply-add. *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05)*. IEEE; 2005:131-139.
9. Lemire D, Kaser O, Kurz N. Faster remainder by direct computation: applications to compilers and software libraries. *Software Pract Exp*. 2019;49(6):953-970.
10. Warren HS. *Hacker's Delight*. 2nd ed. Addison-Wesley Professional; 2013.
11. GNU C Library. time/offtime.c. 2020. https://tinyurl.com/yyr7uazb
12. Microsoft.NET. DateTime.cs. 2020. https://tinyurl.com/y4kej3mmsrc/libraries/system.private.corelib/src/system/datetime.cs
13. Baum P. Date algorithms. 1998. https://tinyurl.com/y44rgx2j
14. Fliegel HF, Flandern TC. Letters to the editor: a machine algorithm for processing Calendar dates. *Commun ACM*. 1968;11(10):657-658.
15. Hatcher DA. Simple formulae for Julian day numbers and Calendar dates. *Q J R Astron Soc*. 1984;25(1):53-55.
16. Hatcher DA. Generalized equations for Julian day numbers and Calendar dates. *Q J R Astron Soc*. 1985;26(2):151-155.
17. Boost C++ Libraries . include/boost/date_time/gregorian_calendar.ipp. 2020. https://tinyurl.com/y4buxmmf
18. LLVM Project. libcxx/include/chrono. 2019. https://tinyurl.com/yytw67zb
19. Reingold EM, Dershowitz N. *Calendrical Calculations: The Ultimate*. 4th ed. Cambridge University Press; 2018.
20. GNU C Library. time/offtime.c. https://tinyurl.com/y42pkbhp
21. OpenJDK. LocalDate.java. jdk/src/java.base/share/classes/java/time/LocalDate.java, 2019. https://tinyurl.com/y92svzxw
22. Android. ojluni/src/main/java/java/time/LocalDate.java. 2017. https://tinyurl.com/ycsltdnq
23. Zeller C. Die Grundaufgaben der Kalenderrechnung auf neue und vereinfachte Weise gelöst. *Württ Vierteljahrsh Landesgesch*. 1882;5:313-314.
24. Troesch A. Droites discrètes et calendriers. *Math Sci Humaines*. 1998;141:11-41. doi:10.4000/msh.2760
25. Neri C, Schneider L. Supplementary Material for Euclidean Affine Functions and their Application to Calendar Algorithms. 2022. https://github.com/cassioneri/eaf
26. GNU Compiler Collection. GCC 11 Release Series, Changes, New Features, and Fixes. 2021. https://tinyurl.com/475dy6vx
27. Linux Kernel. drivers/rtc/lib.c. 2021. https://tinyurl.com/mr764jm5
28. Linux Kernel. kernel/time/timeconv.c. 2021. https://tinyurl.com/4hx3bzaw
29. Google. Benchmark v1.5.2. 2020. https://tinyurl.com/y29mh7q5

How to cite this article: Neri C, Schneider L. Euclidean affine functions and their application to calendar algorithms. *Softw Pract Exper*. 2023;53(4):937-970. doi: 10.1002/spe.3172

[Correction added on 21 December 2022, after first online publication: the format of the references were corrected in this version.]