Algorytmy

Teaching TEX trigonometry and fractions

Bogusław Jackowski, BOP s.c.

Introduction. Surprisingly, Donald E. Knuth did not equipped TEX with trigonometric operations. Math-oriented users, however, would gladly compute $\sin(\alpha)$ or $\cos(\beta)$ from time to time... Fortunately, Knuth published all clues sufficient to make TEX an expert in trigonometry—they can be found in the source of METAFONT. This sketch was inspired by the captivating lecture of `mf.web`.

The problem is that the only floating point operation provided by TEX (or more adequately: pseudo-floating point) is "dimen coercing." You may write something like `3.14159\diameter`, where `\diameter` is a dimen register, in order to compute the perimeter of a circle. However, the accuracy of coercing is hardly satisfactory. Assume, for example, that a dimen register `\q` contains the value 1000000 scaled points (ca 15 points). The assignment `\q=1.000007629\q` does not influence the contents, while after the assignment `\q=1.000007630\q` the contents grows suddenly by 15 scaled points. Needless to say, this eliminates coercing as a proper TEXnique for floating point calculations. In particular, summing Taylor's series using coercing does not seem appropriate.

It turns out that trigonometric calculations can be carried out efficiently using fixed point arithmetic. How? This is the main subject of this sketch.

Computing using pre-computing. There is and old programmer's trick known as pre-computing: some crucial quantities, involved in the process of computation, are computed separately. The results of pre-computing are used in the main algorithm in order to improve its efficiency.

For example, coefficients of a given polynomial of degree n can be transformed in such a way, that the number of multiplications necessary for computing the value of this polynomial can be at most $\lfloor n/2\rfloor + 2$, while the renowned Horner's scheme requires n multiplications. Such pre-computing is usually time- and space-consuming, but is carried out only once. It makes sense only if a pre-computed formula is to be used heavily. This is precisely the case with trigonometric functions.

Trigonometric recollections. Let's recall elementary trigonometric dependencies:

$$\sin(\alpha+\beta)=\sin(\alpha)\cos(\beta)+\sin(\beta)\cos(\alpha),\qquad \cos(\alpha+\beta)=\cos(\alpha)\cos(\beta)-\sin(\alpha)\sin(\beta). \tag{1}$$

The clue is that the value of a sine and a cosine of a sum can be expressed in terms of these functions applied to summands. Therefore, if we can split an arbitrary argument into a sum of specifically chosen summands, we can pre-compute the values of sine and cosine for these summands and hence we can compute the value of a sine or cosine for this argument using iteratively formulas (1).

Coins, powers of two and trigonometry. The problem of evaluating trigonometric functions using pre-computed values is similar to the "problem of coins": assume that there is a collection of coins having values, in certain units, $v_1 = 50$, $v_2 = 20$, $v_3 = 10$, $v_4 = 5$, $v_5 = 2$, and $v_6 = 1$. The question is: how to pay a sum of u units using these coins? Everybody solves such problems every day without any trouble. Therefore, we can expect that the evaluation of trigonometric functions is equally simple. It is, indeed.

The problem of coins can formally be stated as follows: given a finite sequence of positive integers $v_1 \geq v_2 \geq \ldots \geq v_n$ (the ordering is not essential, but convenient), and a non-negative integer u, find non-negative integer coefficients $a_1, a_2, \ldots, a_n$ such that:

$$u = a_1v_1 + a_2v_2 + \ldots + a_nv_n. \tag{2}$$

The algorithm is self-suggesting:

algorithm: THE PROBLEM OF COINS
given: amount of money u,
$v_1 \geq v_2 \geq \ldots \geq v_n$ (coins; any number of specimens of each coin is available—sigh!);

result: a money-bag V containing coins, $\sum_{v \in V} v = u$.

$k \leftarrow 1$; $r \leftarrow u$; $V \leftarrow$ empty
repeat { **invariant:** $r + \sum_{v \in V} v = u$ }
  if $(v_k \leq r)$
    put the coin of value v_k into V; $r \leftarrow r - v_k$
  else $k \leftarrow k + 1$
  fi
while $(r > 0$ **and** $k \leq n)$

The algorithm halts for an arbitrary data, but the money-bag contains the requested set of coins only if r equals zero on exit. In general, the solution to the "problem of coins" is not unique, it may even not exist, as everybody's experience shows, if the set of coin values is not rich enough. In our case, the presence of the coin of value 1 guarantees that the algorithm produces the proper output. For this particular set of coin values, the algorithm finds the solution that requires the smallest number of coins. In general, this is not necessarily true.

A "real programmer" would certainly prefer the set of coins representing powers of two: $v_1 = 64$, $v_2 = 32$, $v_3 = 16$, $v_4 = 8$, $v_5 = 4$, $v_6 = 2$, $v_7 = 1$. From the point of view of the algorithm formulated above, however, the change would be not essential: for $0 \leq u \leq 127$ the solution would be unique and, moreover, the coefficients a_i would be either 0 or 1.

The main difference between the problem of coins and trigonometric calculations is that the latter involves fixed point fractions rather than integers, that is, the "least coin" is 2^{-k}, for some positive integer k, rather than 1. In other words, numbers being considered are multiplicities of 2^{-k}.

One can expect that in practical trigonometric applications 30, 45, and 60 degrees will frequently appear. Hence, an appropriate collection of "angle coins" should contain these values. Negative values and values equal to or greater than 90 degrees can be eliminated by trigonometric identities which implies that the remaining values should be less than 30 degrees. Powers of two, omnipresent in computer applications, can be used for this purpose. All in all, the following set of angles seem adequate:

$$v_1 = 60,\ v_2 = 45,\ v_3 = 30,\ v_4 = 16,\ v_5 = 8,\ v_6 = 4,\ v_7 = 2,\ v_8 = 1,\ v_9 = 2^{-1},\ \ldots,\ v_n = 2^{-n+8}, \quad (3)$$

where 2^{-n+8} corresponds to the maximal accuracy of fixed point arithmetic. Which n should we use in TeX? We'll come back to this issue later.

The problem of coins can be translated into the problem of the evaluation of trigonometric functions as follows: given the sequence $v_1, v_2, \ldots, v_n$ defined as in (3), find non-negative integer coefficients a_1, a_2, $\ldots$, a_n such that

$$\alpha = a_1 v_1 + a_2 v_2 + \ldots + a_n v_n, \quad (4)$$

for $\alpha = k\, v_n$, where k is a non-negative integer.

The algorithm for solving the problem of coins can easily be adapted to the evaluation of trigonometric functions—instead of completing a set of coins in a money-bag, we will calculate iteratively the value of sine and cosine:

algorithm: THE EVALUATION OF SINE AND COSINE
given: α (angle; without the loss of generality, it can be assumed that $0 \leq \alpha < 90$),
$v_1, v_2, \ldots, v_n$ (every v_i is a multiplicity of v_n, v_n is the smallest available number),
$s_k = \sin(v_k)$, $c_k = \cos(v_k)$, $k = 1, 2, \ldots, n$ (pre-computed values of trigonometric functions);
result: $s = \sin(\alpha)$, $c = \cos(\alpha)$.

$c \leftarrow 1$; $s \leftarrow 0$
$k \leftarrow 1$; $\delta \leftarrow \alpha$
repeat { **invariant:** $s \approx \sin(\alpha - \delta)$, $c \approx \cos(\alpha - \delta)$ }
  if $(\delta > v_k)$
    $t \leftarrow s$

```
6.          s ← s ∗ c_k + s ∗ s_k;   c ← c ∗ c_k − t ∗ s_k
7.          δ ← δ − ν_k
8.      else   k ← k + 1
9.      fi
10. while (δ > 0)
```

The algorithm halts for an arbitrary input: because α is by definition a multiplicity of ν_n, we can be sure that δ will eventually become zero. Therefore, the result is always an approximation of the exact solution. The input data, however, influences the efficiency and the accuracy of computation: both depend on the number of iterations. The number of iterations never exceeds n for the set of predefined angles given by (3) and the input angle in the range $0 \leq \alpha < 167$. In other words, coefficients $a_1, a_2, \ldots, a_n$ occurring in (4) are either 0 or 1 within this range. Note that the initial values of c and s (line 1) need not to be 1 and 0. Starting with an arbitrary vector (c, s) would yield (c, s) rotated by α degrees.

The algorithm can slightly be improved if we know in advance that the coefficients are either 0 or 1, since the operation $k \leftarrow k + 1$ can be performed unconditionally, but this is rather less important technical issue.

Integer fractions. The critical aspect of numeric processing is accuracy. In the algorithm evaluationg sine and cosine, the statements in line 6 require special care. As was already mentioned, TEX "coerced" floating point arithmetic is inexact and thus unsuitable for this purpose.

How to multiply fractions using integer representation? As we will see, it suffices to know how to compute the (rounded) product of an integer and a fraction. The problem is not as difficult as it may look at the first glance—actually, every child in a primary school knows how to do it. Let's look at the following scheme (hopefully, no explanations concerning the involved procedure are needed):

$$\begin{array}{r}
27 \\
\times\ 0.8125 \\
\hline
135 \\
54 \\
27 \\
216 \\
\hline
21.9375
\end{array}$$

Observe that the middle part of the diagram *does not contain periods*, that is, the computation, in fact, is carried out on integers. Thinking in terms of binary representation is usually more adequate for programming. Obviously, the main idea of the procedure of multiplication does not change:

$$\begin{array}{rl}
11011 & \\
\times\ 0.1101 & \\
\hline
11011 & \\
00000 & \\
11011 & \\
11011 & \\
\hline
= 10101.1111 & \approx 10110
\end{array}$$

We are interested in an integer result, hence the final rounding. Note that using manual multiplication we obtained the result with the accuracy $\pm\frac{1}{2}$.

For programming purposes, it is more convenient to accumulate the partial result rather than to collect all intermediate products. The modified procedure is shown below (the accumulated partial result is underlined; adding 1 in the final step accomplishes rounding):

$$\begin{array}{rcr}
11011 & & \\
\times\ 0.1101 & & \\
\hline
(\underline{0} + 11011)\,\mathbf{div}\,2 & \rightarrow & 1101 \\
(\underline{1101} + 00000)\,\mathbf{div}\,2 & \rightarrow & 110 \\
(\underline{110} + 11011)\,\mathbf{div}\,2 & \rightarrow & 10000 \\
(1 + \underline{10000} + 11011)\,\mathbf{div}\,2 & \rightarrow & 10110 \\
\hline
& & = 10110
\end{array}$$

This routine is suitable for implementing in TeX, since we can easily check the least binary digit of an integer—if the number is odd, the digit is 1, otherwise it is 0. The modified routine can be stated in an algorithmic form as follows:

algorithm: THE MULTIPLICATION OF AN INTEGER BY A "SCALED" FRACTION
given: positive integer p and fraction f, the latter represented by an integer h such that
$f = h/2^{k^*}$ for some positive integer k^* (2^{k^*} is the representation of 1);
result: integer $r = \lfloor pf + \frac{1}{2} \rfloor$.

1. $r \leftarrow 0;\ \ i \leftarrow k^*$
2. **repeat**
3.   **if** $(h$ **mod** $2 = 1)$ $r \leftarrow r + p$ **fi**
4. $i \leftarrow i - 1;$ **if** $(i = 0)$ $r \leftarrow r + 1$ **fi**
5. $r \leftarrow \lfloor r/2 \rfloor;\ \ h \leftarrow \lfloor h/2 \rfloor$
6. **while** $(h > 0)$
7. **if** $(i > 0)$
8.   **repeat**
9. $i \leftarrow i - 1;$ **if** $(i = 0)$ $r \leftarrow r + 1$ **fi**
10. $r \leftarrow \lfloor r/2 \rfloor$
11.   **while** $(i > 0)$
12. **fi**

The algorithm is admittedly ugly. Its main advantage is the straightforward relevance to the manual routine; its main disadvantage is that the rigorous proof of the correctness would be ugly as well.

Much more elegant solution to the same problem can be found in `mf.web` (sections 109–111): simpler, although not necessarily obvious, and susceptible to the proving of correctness. After thinking a while, however, you should recognise the already presented algorithm in it: rounding (lines 4 and 9) is cleverly moved to the initialisation of the variable r and the additional loop (dealing with leading zeros, lines 8–11) is avoided by adding 2^{k^*} (that is, 1) to h at the very beginning. Below, a slightly modified version of the Knuthian solution is demonstrated:

algorithm: THE IMPROVED MULTIPLICATION OF AN INTEGER BY A "SCALED" FRACTION
given: positive integer p and fraction f, the latter represented by an integer h such that
$f = h/2^{k^*}$ for some positive integer k^* (2^{k^*} is the representation of 1);
result: integer $r = \lfloor pf + \frac{1}{2} \rfloor$.

1. $h \leftarrow h + 2^{k^*}$
2. $r \leftarrow 2^{k^*-1}$
3. **repeat**
4.   **if** $(h$ **mod** $2 = 1)$ $r \leftarrow (r + p)$**div** 2
5.   **else** $r \leftarrow r$ **div** 2
6.   **fi**
7. $h \leftarrow h$ **div** 2
8. **while** $(h > 1)$

The proof of the correctness of the "nicer" version is elementary. Nevertheless, it is worthy of detailed presentation—it is nice to know that the correctness of the basic algorithms of METAFONT can be formally proved, isn't it?

We will prove that the following invariant holds at line 3, that is, at the begining of the loop:

$$\lfloor (ph + r)/2^k \rfloor - p = \lfloor ph_0/2^{k^*} + \tfrac{1}{2} \rfloor, \quad 2^k \le h < 2^{k+1}, \tag{5}$$

for some non-negative integer k, where h_0 is the value of variable h on the entry to the algorithm. The invariant obviously holds, for $k = k^*$, after performing the assignments in lines 1 and 2.

In the sequel, a prime will denote the value of a variable after an assignment. If h is odd, the assignment in line 4 is performed and we have $r + p = 2r' + (r + p)\,\mathbf{mod}\,2$, that is, $r = 2r' - p + (r + p)\,\mathbf{mod}\,2$;

otherwise, the assignment in line 5 is performed and, similarly, we have $r = 2r' + r \,\mathbf{mod}\, 2$. After the assignment in line 7, we have $h = 2h' + 1$ for odd h and $h = 2h'$ otherwise. Thus, for odd h we have:

$$\begin{aligned} \lfloor (ph+r)/2^k \rfloor &= \lfloor (p(2h'+1) + 2r' - p + (r+p)\,\mathbf{mod}\, 2)/2^k \rfloor \\ &= \lfloor (2ph' + 2r' + (r+p)\,\mathbf{mod}\, 2)/2^k \rfloor \\ &= \lfloor (ph' + r' + \tfrac{1}{2}((r+p)\,\mathbf{mod}\, 2))/2^{k-1} \rfloor \\ &= \lfloor (ph' + r')/2^{k'} \rfloor, \end{aligned} \tag{6}$$

where $k' = k-1$; the term $\frac{1}{2}((r+p)\,\mathbf{mod}\, 2)$ can be neglected inside a 'floor' operation because both $ph' + r'$ and $2^{k'}$ are integers, while $\frac{1}{2}((r+p)\,\mathbf{mod}\, 2) \le \frac{1}{2}$. Similarly, for even h we have:

$$\begin{aligned} \lfloor (ph+r)/2^k \rfloor &= \lfloor (p(2h') + 2r' + r \,\mathbf{mod}\, 2)/2^k \rfloor \\ &= \lfloor (2ph' + 2r' + r \,\mathbf{mod}\, 2)/2^k \rfloor \\ &= \lfloor (ph' + r' + \tfrac{1}{2}(r \,\mathbf{mod}\, 2))/2^{k-1} \rfloor \\ &= \lfloor (ph' + r')/2^{k'} \rfloor. \end{aligned} \tag{7}$$

Therefore, the value of the expression $\lfloor (ph+r)/2^k \rfloor - p$ remains constant after every iteration. Obviously, $h = 1$ (that is, $k = 0$) after the final iteration and the left part of equation (5) reduces simply to $\lfloor p + r \rfloor - p = r$, which eventually completes the proof:

$$r = \lfloor ph_0/2^{k^*} + \tfrac{1}{2} \rfloor = \lfloor pf + \tfrac{1}{2} \rfloor. \tag{8}$$

Practical numerology. In order to implement in TEX the algorithms outlined above we have to answer a question how to represent angles and the respective values of sine and cosine.

The smallest TEX unit is a *scaled point*. The following dependencies define its relation to basic TEX units: 1 point $= 2^{16}$ scaled points, 1 inch $= 72.27$ points. The maximal dimension that TEX handles smoothly is `\maxdimen` $= 2^{30} - 1$ scaled points ≈ 16383.99998 points.

It makes sense to assume that angles of order 360 degrees can appear in practical applications. Engineers have a good custom of inventing "safety coefficients." Borrowing their custom, we can introduce a "safety coefficient" equal to 2.8444... and assume that, from a practical point of view, $360 \times 2.8444\ldots = 1024$ is the largest acceptable value. This implies that 16 points can be accepted as the representation of 1 degree. Coming back to equation (3), this means that we can set $n = 20$. Of course, an input angle can (and should) be reduced to the first octant using elementary trigonometric identities.

The relevant values of sine and cosine can be represented with higher accuracy, because the values of sine and cosine for angles in the first quadrant fall within the range $[0, 1]$. If they were represented in TEX as dimensions (that is, stored in *dimen* registers), they would be bounded by $\le$ `\maxdimen` $= 2^{30} - 1$; storing them in *count* registers allows to use values up to $2^{31} - 1$. Therefore, implementing them as counts and using the value 2^{30} as the representation of 1 seems reasonable. In other words, we can set $k^* = 30$ in the algorithms computing the product of an integer and a fraction.

Eventually, we can complete the table of sine and cosine values we can use in TEX:

index	α	$2^{20}\alpha$	$\lfloor 2^{30}\cos(\alpha) + \frac{1}{2} \rfloor$	$\lfloor 2^{30}\sin(\alpha) + \frac{1}{2} \rfloor$
1	60	62914560	536870912	929887697
2	45	47185920	759250125	759250125
3	30	31457280	929887697	536870912
4	16	16777216	1032146887	295963357
5	8	8388608	1063292242	149435979
6	4	4194304	1071126243	74900443
7	2	2097152	1073087729	37473049
8	1	1048576	1073578288	18739379
9	1/2	524288	1073700939	9370046
10	1/4	262144	1073731603	4685068

11	1/8	131072	1073739269	2342539
12	1/16	65536	1073741185	1171270
13	1/32	32768	1073741664	585635
14	1/64	16384	1073741784	292818
15	1/128	8192	1073741814	146409
16	1/256	4096	1073741822	73204
17	1/512	2048	1073741823	36602
18	1/1024	1024	1073741824	18301
19	1/2048	512	1073741824	9151
20	1/4096	256	1073741824	4575
21	1/8192	128	1073741824	2288
22	1/16384	64	1073741824	1144
23	1/32768	32	1073741824	572
24	1/65536	16	1073741824	286
25	1/131072	8	1073741824	143
26	1/262144	4	1073741824	71
27	1/524288	2	1073741824	36
28	1/1048576	1	1073741824	18

This table and the presented algorithms have been employed in the TRANS package that facilitates using PostScript transformations in plain TEX. Readers interested in technical details of the implementation can find the package at URL:

`ftp://ftp.GUST.org.pl/TeX/macros/generic/TeX-PS/trans.tex`

Sending comments to `bop@bop.com.pl` is welcome. ☐

◇ Bogusław Jackowski, BOP s.c.