COVER STORY

Pope Gregory on Mars on February 31, 2000

—by *Clement Kent*
Toronto, Ontario, Canada

IT IS STILL HARD, even at this very late date, to convince some businesspeople and programmers that the Year 2000 bug is a serious problem. It is still harder to convince skeptics that Y2K bug fixing and testing should include the leap day period around February 29, 2000. Many test plans focus on the rollover from December 31, 1999 to January 1, 2000, where the largest number of bugs will be concentrated. Expanding tests to include Monday, February 28 2000 to Wednesday March 1 2000, or even to Sunday March 5 2000, can be a hard sell.

In this article I want to persuade you that the leap day period is a time of serious risk to computer systems, and to discuss algorithms for dealing correctly with dates over a time span of more than 4000 years. Although correct implementation of date conversion routines is essential to avoiding the leap day year 2000 bug, there is another reason for looking into the code: it will lead us into some thoughts on generality, clarity, and efficiency in APL coding styles which may be of use to the educational community.

Why worry about the Leap Day?

The most important business issue about leap-day testing and bug fixing is simple: "why spend the money on this when we have serious Jan. 1 2000 problems to fix?" This is not a rhetorical question—I have heard it expressed by Y2K project managers and technical people who have to make difficult decisions about where to allocate scarce people, money, and machine resources on their Y2K projects. Their argument usually goes "Leap-day bugs are so rare compared to century bugs that, given we can't achieve 100% code coverage in our testing, it makes sense to focus on the century bugs."

This is a fair question. In mainframe systems, it is often costly and time-consuming to arrange for extra testing time at future dates. Writing test plans and building test data sets and test drivers is as big a task for a week in February-March as it is for a week in December-January. If most of the costs are fixed, but the risks to be avoided are smaller, why spend the money on leap-day tests?

Illustration created by Jon McGrew. Photo of Mars landscape provided through the courtesy of NASA. Used with permission.

The single best answer I can give to this "Why do it?" question is "*because systems will fail, sometimes catastrophically*." As computer professionals, we are obliged to minimize our clients' exposure to rare risks if these may involve catastrophic failures. (*Ref's 1,2*). Leon Kappelman has said "I have noticed an unhealthy tendency...in some Year/2000 circles...to belittle or criticize anyone who takes seriously Year 2000 risks that can be classified as high magnitude (impact), but low probability (in occurrence)."(*Ref. 2)*

Anyone who doubts that missing a leap day can be catastrophic will profit from reading the description in *Time Bomb 2000 (ref 3)* of the New Zealand and Australian aluminum smelters which suffered major damage just after midnight on February 29, 1996, due to crashes in the real-time microcontrollers which managed the heating of the crucibles. This well-documented example should give us pause, for it shows that millions of dollars of damage can be caused by a simple leap-day bug in programs.

My second answer to "Why do it?" is "*because the leap-day problem is less well known than the century bug*." Those who are well-informed about calendrical rules and date calculations may be surprised how incompletely understood these are, even among programmers. If everyone stuck to the "known-good" public date utilities in their code, this wouldn't be a problem. But, the apparent simplicity of the leap year rules combined with the do-it-yourself tendency of APL code cowboys often results in hand-tailored leap-year calculations inside of larger functions. If the programmer was incompletely informed, bugs result.

A little knowledge is a dangerous thing

My favorite way of convincing people that the leap-day problem is real is to discuss it in a large group of APL professionals. Each time I have done so, I have found at least one or two very experienced programmers willing to assert that 2000 was *not* a leap year. I guesstimate the incidence of code-cowboyism at around 50% in the APL-using community, which means about half of those who were wrong could have embedded bad leap year logic into their code.

And indeed, we *have* seen hand-coded incorrect leap year logic in actual use in large APL systems in the course of our Year 2000 work. The incidence is not enormous, but it is high enough for me to suggest that many APL applications that meet the following criteria will have leap-day problems in 2000 unless fixed:

- The application uses date logic
- There were no coding standards to encourage use of common utilities
- The application is large

Given that we know the problem exists and may effect the systems we use, and given that funding is at hand anyways for code fixing and testing, I assert it is unprofessional not to plan for leap year fixing in your Y2K project.

What are the leap year rules?

The rules to use are simply stated:

1. If 4000 evenly divides the year, it is *not* a leap year
2. If 400 evenly divides the year, but 4000 doesn't, it *is* a leap year
3. If 100 evenly divides the year, but 400 doesn't, it *is not* a leap year
4. If 4 evenly divides the year, but 100 doesn't, it *is* a leap year
5. [If 1 evenly divides the year, but] 4 doesn't, it *is not* a leap year.

Beyond the 4000 year rule, simple algorithms don't work—over a period of 20,000 years the length of the year changes enough that new rules are required.

These rules boil down nicely to this line of APL, which will yield 1 if "year" is a leap year:

```
∨≠(0≠4000 100∘.|year)∧ 0=400 4∘.|year←⌊year
```

These rules, then, are sufficient to correctly calculate all dates between 1 AD to past 10,000 A.D.—a range sufficient for most business purposes! Interestingly, most utilities from major APL vendors handle rules 2-5 correctly but ignore rule 1, on the principle I think that the authors won't be alive when the functions break. A second reason (to be explored later in this paper) is that implementing the 4000 year rule adds slightly to the computational cost of the utility routines. Both of these reasons are similar in nature to the reasoning that got us into the Y2K mess in the 70's and 80's. I think that for the sake of completeness and professional pride, common date utilities should handle the full set of leap year rules.

Although the unfortunate programmer of the NZ smelter code forgot even the simplest 4 year rule, it is not uncommon to find people who are unaware of the 100 year rule. Although this invalidates calculations for 1900 and 2100, it is quite fortunately painless in 2000, since the 4 year rule alone gives the same result as applying the full rule set.

The danger comes from that half-knowledgeable cowboy who knew about the 4 and 100 year rules, but not the 400 year rule. This cowboy's bronco will buck in the year 2000, asserting it is not a leap year, just like 1900 and 2100. This is the error to worry about! (Presumably, those who didn't even put in the four year rule got caught in 1996...)

Where did the rules come from?

The Julian calendar (still used in Russian Orthodox holidays) ran from the time of Julius Caesar up to the time of Pope Gregory XIII in 1582. The Julian calendar used only the 4 year rule. As the year is a bit less than 365.25 days, (actual length: 365 days, 5 hours, 48 minutes, 46 seconds) the Julian calendar date slowly fell behind the "true" date. The error accumulated at about 3 days per 400 years.

The shift in calendar date versus true date might not have been noticed were it not for the Christian religion's dependence on the Jewish religion's lunar calendar rules. Easter and Passover are linked, and Passover is determined from the night of the first full moon of the first month of spring. Thus, the date of the spring equinox is important in calculating the date of Easter, the central Christian holy day.

The date of the equinox can be independently determined by astronomical measurements. By 1586 these revealed the equinox was arriving near March 11, rather than on March 21 as it had in 325 AD when the Council of Nicaea codified the rules for the date of Easter. Pope Gregory called upon a council of wise men to devise a correction to the Julian calendar, so that the date of Easter might be stabilized. The recommendations of this council gave rise to the Gregorian calendar.

As we worry about the year 2000's computer woes, give a thought to poor Pope Gregory. His advisors dictated that 10 days be added to the calendar date to retrieve the ancient date of the equinox. October 4 1582 was followed by October 15. Imagine the havoc this would wreak in computer systems today! Riots ensued in several parts of the globe as simple folk objected to a Papish plot to rob them of ten days of their lives. The implementation date of the Gregorian reforms was delayed until 1752 in England and 1923 in Greece.

The Gregorian reform added the 100 and 400 year rules. More accurate measurements have since added the 4000 year rule.

How were the rules determined? ...a mathematical diversion

A long tradition, going back to the ancient Greeks and before, expressed numbers as sums of fractions. The year length is 365 plus:

```
      (t⊥5 48 46)÷×/t←24 60 60
0.242199074074074
```

Pope Gregory's approximation was (¼ – ⅟₁₀₀) + ⅟₄₀₀ = 0.2425, which is not bad—only about .0003 off. Adding the modern correction gives:

```
      –/÷4 100 400 4000
0.24225
```

which differs from the actual value by 4.4 seconds, or one day in 19,646 years. Good enough!

At this point the pragmatic person stops: we know enough now. I, however, asked myself "why 4 and 100 and 400? What other values could be chosen?" This seemingly pointless question actually helped me improve my understanding of some practical algorithms, as we shall see.

A good starting point is to ask "can we express any real number between 0 and 1 as the sum of a series in which the terms are reciprocals of integers?" Teachers of first-year calculus may wish to point out to students the fact that the sum –/÷⍳*n* is conditionally convergent, and so can be rearranged to converge to any real number. So we can answer our question with a yes if we accept conditionally convergent series.

If we ask whether an absolutely convergent series of reciprocals of integers can be found, we can give a constructive answer which may challenge the student a bit to prove. Let *X* be a number in the open interval (0,1), and define by recursion the vector *N*←*N*,⌊.5+÷*X*–+/÷*N*

```
[0] N←epsilon Approx1 X
[1] ⍝ give a vector N of integers such that epsilon ≥ | X – +/÷N
[2] ⍝ X ∊ (0,1)
[3] N←⍳0
[4] :WHILE (epsilon < |X)
[5]      X←X–÷¯1↑N←N,⌈¯0.5+÷X
[6] :ENDWHILE
```

Question for the student: how fast will the series +/÷*N* converge to *X*? (Proving the *minimum* rate of convergence is not too hard. Proving the *average* rate requires more advanced concepts!)

Boole rules!

Try this algorithm with the fractional part of the actual length of the year (using the same epsilon the 4000 year rule achieves):

```
      .00005 Approx1 .24219907
4 ¯128
```

I like to think of this new, revised rule as Boole's reform of the calendar. Boole's rules are simple: if the year is divisible by 128, it's not a leap year, else if it's divisible by 4, it is. In two terms, ideally suited for the digital computer, it converges to the actual year length better than the four terms of the modified Gregorian rule. Furthermore it can be implemented by bit logic on the base two representation of the year, making even the C and Assembler programmers happy!

Unfortunately, I have received no reply to my suggestion to the Vatican that they decree a switch to Boole's rule in the year 2048.

One strike against Boole's rule is that it is oriented to base 2. If only people had four fingers on each hand! I resolved to try again with a different algorithm:

```
[0] N←epsilon Approx2 X
[1] N←⍳0
[2] →⎕LC × epsilon<X←X−÷¯1↑N←N,⌈÷X

      .00005 Approx2 .24219907
5 24 1879
```

The careful reader will note that approximation 2 gives a series of positive terms which approach X from below, rather than sometimes oscillating around X the way the first algorithm might. The above rule implies that we have leap days in years divisible by 5, 24, and 1879. How many leap days do we have in years divisible by 120? Unfortunately, the 1879 in the result convinced me that some kind of rounding off needs to be applied to make the result usable by ordinary "count on the fingers" human calculators.

```
[0] N←epsilon Approx3 X;last
[1] N←0 ⋄ last←1
[2] →⎕LC × epsilon<X←X−÷¯1↑N←N,last←last×⌈÷X×last

      .00005 Approx3 .24219907
5 25 475 10925
```

This third approximation is a bit better. It forces the i+1st term in the vector N to be a multiple of the ith term. If we round off the last terms of this rule we get the rule `5 25 500 5000`, which is pleasingly simple to evaluate for people and quite accurate. Of course, this "rule of fives" always adds leap days when the higher order terms kick in, so 1970 had 1 leap day, 1975 had 2, and 2000 would have 3. We truly would be able to celebrate the millennium on February 31, 2000, if only we used this rule!

This digression shows us one goal Gregory's advisors had—to make sure there was always at most one leap day in a year. This is desirable to ensure the calendar never gets more than one day out of step with the real world. Thus, we'd like to find a vector N such that odd-numbered elements of N are positive and even-numbered elements negative, `N[I+1]` is always a multiple of `N[I]`, and `+/÷N` converges to the target X. The negative elements of N can be interpreted as "don't have leap years when the year is divisible by `N[I]`", as in "years divisible by 100 are not leap years."

```
[0] N←epsilon Approx4 X;last
[1] ⍝ give a vector N of integers such that epsilon ≥ | X − +/÷N
[2] ⍝ where X ∊ (0,1); 0=N[i]|N[i+1] ;  0>N[2×i] ; 0<N[1+2×i]
[3] N←⍳0 ⋄ last←1
[4] N←N,last←(×X)×(|last)×⌊|÷X×last ⋄ →⎕LC×epsilon < |X←X−÷last

      .000005 Approx4 .24219907
4 ¯128 86400
```

As a *very* challenging exercise, ask your students to prove how quickly the series delivered by `Approx4` converges. As the example below where X=1/e shows, the "nice to use" algorithm can sometimes be quite slow to converge, compared to the earlier algorithms:

```
      .000005 Approx4 *−1
2 ¯6 24 ¯120 720 ¯5040 40320
      .000005 Approx3 *−1
3 30 840 45360
      .000005 Approx1 *−1
3 29 15786
```

What have we learned from this digression? We now have a computationally sound way to determine leap year rules on any planet (exercise for the student: look up day length and year length on Mars and determine what Pope Gregory might have done had he been Martian). We understand why the alternating sum (e.g. `−/÷4 100 400 4000`) is preferred to the normal sum (e.g. `+/÷5 25 500 5000`). And now we are ready to recast some of the leap year algorithms that have practical use so they are simpler and more Martian-friendly.

Real code: Is this a leap year?

An example of this new generality is our "is this a leap year?" codelet:

```
(×N)+.×0=N∘.|years
```

This lovely little phrase assumes N is a rules vector of the kind produced by `Approx4`, above, such as `N←4 ¯100 400 ¯4000`. It will work on Mars or Earth, and with any rank for `years`. Compare it to the APL2000 `LEAPYR` function:

```
[0] R←LEAPYR A;⎕IO
[1] ⎕IO←⍳0
[2] R←((100⍴ 1 0 0 0),300⍴ 0 0 0 0 ,96⍴ 1 0 0 0)[400|A⌈1]
```

It may take you a moment, but eventually you'll convince yourself this is the `4 100 400` rule set. Like most of the APL2000 date functions (workspace `DATES`), it has been optimized for performance, not for readability. It is not fun to imagine rewriting it for Martians, let alone the 4000 year rule. It will, however, be faster than our elegant morsel, if you don't need the 4000 year rule, and it uses less memory.

Real code: Julian day numbers

A common problem of considerable importance in Y2K projects is to convert a (Y M D) date triplet (such as is obtained from `⎕TS`) into a day number. Day numbers are usually defined as the number of days since a fixed date in the past. Astronomers use a particular fixed date and call their values "Julian day numbers." In Sharp APL the corresponding function (which uses Year 0 as the base) is called "`tojul`", in honor of the astronomical lingo.

An inverse function goes from the day number back to a three-element Y M D array. In Sharp APL this is called "`togreg`". The corresponding APL2000 functions are *DATEBASE* and *DATEREP*.

These functions are very useful for determining time spans, time comparisons, and for finding new dates. The time span between two dates *YMD1* and *YMD2* is:

```
(tojul YMD2)- tojul YMD1
```

Tomorrow's date is:

```
togreg 1+tojul 3⍴⎕TS
```

To my chagrin, when I first read the code for the public versions of these functions, I was utterly baffled. I couldn't tell whether they implemented the 100 year rule, the 400 year rule, or none of the above! I set out to create simplified functions that even I, or a Martian, could understand.

My algorithm calls for the year to have "origin March 1." With this correction made, the number of days up to March 1 of "`yr`" is the beautifully simple phrase:

```
-⌿⌊coeff∘.×yr
```

where "`coeff`" is `365 0 0 0 + ÷ 4 100 400 4000`. If using Boole's rule, `coeff` would be `365 0 + ÷ 4 128`. Martians may make their own substitutions.

Compare the above lucid code to the equivalent part of the very efficient APL2000 code from *DATEBASE* (transformed somewhat for readability):

```
((LEAPYR R)∧M<march)-((⌊Y×146097)-(R←400|Y)+(100×4|Y)-4×100|Y)+400
```

If we have a beauty contest, I think my code will win! For a fair timing comparison with vendor functions I wrote the following "`tojulMars`" function:

```
[0]  dno←tojulMars ymd;yr;mo;day;a;coeff;mlen;⎕IO;⎕CT
[1]  ⎕CT←0 ∘ ⎕IO←1
[2]  yr←+/1 0 0/ymd ∘ mo←+/0 1 0/ymd ∘ day← +/0 0 1/ymd
[3]  coeff←365.25 .01 .0025 .00025     ⍝  ÷N←4 100 400 4000
[4]  mlen←306 337 0 31 61 92 122 153 184 214 245 275
[5] ⍝mlen←¯2⌽+\0 31 30 31 30 31 31 30 31 30 31 31
[6]  yr←yr+(mo>2)+¯1 ∘ →(∨/,a←yr<100)↓l1
[7]  yr←yr+a×100×⌊⎕TS[1]+100  ⍝2-digit yr → current century
[8] l1:dno←-⌿⌊coeff∘.×yr       ⍝ the real work is done here
[9]  dno←dno+day+mlen[mo]
```

Although it is messy, this function works and is easy to adapt to Mars. The rule set used is clearly outlined in line `[3]`, while the real work (finding the number of days from the base to March 1 of the year specified) is done in line `[8]`. Timings show it is in some cases faster, in some slower, than the optimized functions from APL2000 and Soliton.

I declare this a victory for Martians everywhere! ■

Julius Caesar I, Pope Gregory 0

Timings of Julian date functions

Timings performed using APL+WIN on a Pentium 100 PC.

- Times shown are thousandths of a second per execution.
- Rankings on other architectures will vary. Timings on Sharp APL Mainframe showed `tojulMars` faster than *DATEBASE*

The contenders

Tojul4only	If we only need to handle 1901-2099, just use the 4 year rule!
DATEBASE	APL2000's optimized version
Tojulv22	Soliton's optimized version for Sharp APL version 22
TojulMars	My Martian algorithm contestant

Size of argument

Valid Until	Function	1	3	10	32	100	316	1000
2099	*Tojul4only*	0.9	0.9	1.0	1.3	2.2	4.7	13.5
3999	*DATEBASE*	1.3	1.4	1.5	1.9	3.3	8.5	17.3
3999	*Tojulv22*	1.2	1.3	1.5	2.1	4.0	13.6	29.7
20000	*TojulMars*	1.1	1.1	1.3	2.0	4.3	11.0	27.0

Comments

- If you only care about dates up to 2099, the simplest rule (`tojul4only`) is fastest.
- The APL+WIN function *DATEBASE* is the winner for large arrays on a PC.
- The function `tojulMars` has the 4000 year rule and is fastest for smaller arrays on the PC and for larger arrays on a mainframe.

References

1. Minimizing Risk and Automated Testing, by Don Estes. *Year/2000 Journal*, March/April 1998

2. Keeping our Eyes on the Prize, by Leon Kappelman. *Year/2000 Journal*, March/April 1998

3. *Time Bomb 2000*, by Ed and Jennifer Yourdon.