Three Problems for Math. 128B Solutions due Mon. 9 Feb. 2004

The following computations are to be programmed in Matlab, Fortran, *C*, Basic or Java using ordinary (not arbitrarily high-precision) floating-point arithmetic. The use of automated algebra systems like Maple or Mathematica or Macsyma or … is allowed only to check your results, not to produce them. Credit is awarded not for those results but for their explanation.

Problem 1: Jean-Michel Muller's Recurrence

Starting with $x_0 := 11/2$ and $x_1 := 61/11$, define in turn x_2 , x_3 , x_4 , …, x_N , … from the

Recurrence Formula $x_{N+1} := 111 - (1130 - 3000/x_{N-1})/x_N$.

Compute x_{34} and x_{340} correct to at least 8 sig. dec. in more than one way, using perhaps diverse computers or calculators. Can you detect that something has gone awry and explain why? What short recurrence gets our values x_{34} and x_{340} very fast and accurately despite roundoff?

Solution 1:

The "correct" results computed *exactly* (with no rounding errors) would be

$$x_{34} = 172198118279409596138998 6301 / 28709387656720510591037 5321$$

$$= 5.99797\ 25216\ 84911\ 60006\ 55307\ 61063\ 78444\ 31896 \ldots \quad \text{and}$$

$$x_{340} = 5.99999\ 99999\ 99999\ 99999\ 99999\ 98802\ 22196 \ldots .$$

But one rounding error, like rounding $x_1 := 61/11 = 5.5454545454…$ to the computer's or calculator's working precision, is enough to change both computed results to 100 . Tabulated below are computed results from nine computers' and calculators' arithmetics, namely …

Basica Single:	24 sig. bits provided by IBM PC Basica in ROM in 1982.
Bascom Single:	24 sig. bits provided by IBM PC Basic Compiler in 1982.
T-Basic Single:	24 sig. bits* provided by Borland's Turbo-Basic on an IBM PC.
HP-97:	10 sig. dec. rounded conventionally, almost.
HP-71B:	12 sig. dec. conforming to IEEE standard 854.
MATLAB 68040	53 sig. bits* from Matlab 5 on a 68040-based Apple Macintosh
T-Basic Double:	53 sig. bits* provided by Borland's Turbo-Basic on an IBM PC.
Basica Double:	56 sig. bits provided by IBM PC Basica in ROM in 1982.
T-Pascal Extended:	64 sig. bits* provided by Borland's Turbo-Pascal on an IBM PC.

* These arithmetics perform on hardware that conforms to IEEE Standard 754 for Binary Floating-Point.

Roughly speaking, the more precise the arithmetic, the larger the values of N where computed values X_N switch convergence from the right limit 6 to the wrong limit 100 . Here "wrong" suggests that someone or something deserves blame. Don't blame the computer's floating-point arithmetic. In its defence, an argument could be advanced that 100 is the *right* limit for the data (61/11 rounded off) actually presented to the computer's arithmetic. The limiting value x_∞ turns out to be a *Discontinuous* function of the initial values x_0 and x_1 ; this singularity is what amplifies roundoff so badly. Still, different arithmetics trace different paths through computed values $X_1 \approx 61/11$, X_2, X_3, …, $X_{34} \approx 100$; these variations cry out for explanation.

Different computations of X_N by J-M. Muller's recurrence

N	Basica Single	Bascom Single	T-Basic Single	HP-97 10 Dec.	HP-71B 12 Dec.	MATLAB 68040	T-Basic Double	Basica Double	T-Pascal Extended
0	5.5	5.5	5.5	5.5	5.5	5.5	5.5	5.5	5.5
1	5.545	5.545	5.545	5.545	5.545	5.545	5.545	5.545	5.545
2	5.590	5.590	5.590	5.590	5.590	5.590	5.590	5.590	5.590
3	5.633	5.633	5.633	5.633	5.633	5.633	5.633	5.633	5.633
4	5.675	5.675	5.675	5.675	5.675	5.675	5.675	5.675	5.675
5	5.713	5.668	5.709	5.713	5.713	5.713	5.713	5.713	5.713
6	5.743	4.941	5.681	5.749	5.749	5.749	5.749	5.749	5.749
7	5.676	-10.56	4.577	5.716	5.781	5.782	5.782	5.782	5.782
8	3.948	160.5	-20.52	4.665	5.798	5.811	5.811	5.811	5.811
9	-41.36	102.2	134.1	-18.73	5.608	5.838	5.837	5.838	5.838
10	119.9	100.1	101.5	137.0	1.772	5.861	5.861	5.861	5.861
11	101.0	100.0	100.1	101.6	-224.7	5.884	5.883	5.882	5.881
12	100.1	100.0	100.0	100.1	108.5	5.936	5.919	5.904	5.899
13	100.0	100	100.0	100.0	100.5	6.534	6.244	5.994	5.914
14	100.0	100	100.0	100.0	100.0	15.41	11.20	7.260	5.925
15	100	…	100	100.0	100.0	67.47	53.02	24.29	5.886
16	100		100	100.0	100.0	97.14	94.74	81.49	5.053
17	…		…	100.0	100.0	99.82	99.67	98.65	-11.76
18				100	100.0	99.99	99.98	99.92	156.6
19				100	100.0	100.0	100.0	100.0	102.2
20				…	100.0	100.0	100.0	100.0	100.1
21					100	100.0	100.0	100.0	100.0
22					100	100.0	100.0	100.0	100.0
23					…	100.0	100.0	100.0	100.0
24						100.0	100.0	100.0	100.0
25						100.0	100.0	100.0	100.0
26						100.0	100.0	100.0	100.0
27						100.0	100.0	100.0	100.0
28						100	100.0	100.0	100.0
29						100	100	100	100.0
30						…	100	100	100.0
31							…	…	100.0
32	…	…	…	…	…	…	…	…	100.0
33	100	100	100	100	100	100	100	100	100
34	100	100	100	100	100	100	100	100	100
N	Basica Single	Bascom Single	T-Basic Single	HP-97 10 Dec.	HP-71B 12 Dec.	MATLAB 68040	T-Basic Double	Basica Double	T-Pascal Extended

Each iteration of Muller’s recurrence can be written

$$\begin{bmatrix} x_{n+1} \\ x_n \end{bmatrix} := F\left(\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix}\right) \quad \text{wherein} \quad F\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) := \begin{bmatrix} 111-(1130-3000/y)/x \\ x \end{bmatrix} .$$

The iteration appears to converge to a fixed-point $\mathbf{z} = F(\mathbf{z})$. Every such fixed-point has the form $\mathbf{z} = \begin{bmatrix} z \\ z \end{bmatrix}$ in which z satisfies $z^3 - 111z^2 + 1130z - 3000 = (z-100)(z-6)(z-5) = 0$, so there are three fixed-points $\begin{bmatrix} 100 \\ 100 \end{bmatrix}$, $\begin{bmatrix} 6 \\ 6 \end{bmatrix}$ and $\begin{bmatrix} 5 \\ 5 \end{bmatrix}$. Two are repulsive, one attractive. Which? If a column $\mathbf{v}$ is close to a fixed-point $\mathbf{z}$, then $F(\mathbf{v})-\mathbf{z} = F(\mathbf{v}) - F(\mathbf{z}) \approx F'(\mathbf{z})(\mathbf{v}-\mathbf{z}) + O(\mathbf{v}-\mathbf{z})^2$ where

$$F'\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} (1130-3000/y)/x^2 & -3000/(y^2 \cdot x) \\ 1 & 0 \end{bmatrix}$$

is the *Jacobian* matrix of first partial derivatives of F . Its eigenvalues tell us whether a fixed-point is attractive or repulsive. The eigenvalues μ of $F'\left(\begin{bmatrix} 100 \\ 100 \end{bmatrix}\right) = \begin{bmatrix} 11/100 & -3/1000 \\ 1 & 0 \end{bmatrix}$, satisfying $\det(\mu I - F') = (\mu - 11/100)\mu + 3/1000 = (\mu - 1/20)(\mu - 1/50) = 0$, are both much smaller than 1 in magnitude, so iterates x_n converge rapidly towards 100 when they get close. On the other hand, the eigenvalues μ of $F'\left(\begin{bmatrix} 5 \\ 5 \end{bmatrix}\right) = \begin{bmatrix} 106/5 & -24 \\ 1 & 0 \end{bmatrix}$ are 20 and 6/5 , both rather bigger than 1 , so iterates x_n flee from 5 if ever they come near it. Finally $F'\left(\begin{bmatrix} 6 \\ 6 \end{bmatrix}\right) = \begin{bmatrix} 35/2 & -125/9 \\ 1 & 0 \end{bmatrix}$ has eigenvalues $\mu = 5/6 < 1$ and $\mu = 50/3 > 1$, which implies that iterate pairs $\{x_n, x_{n-1}\}$ can converge slowly towards $\{6, 6\}$ only from a special direction. If iterate pairs depart even slightly from this direction they will flee from $\{6, 6\}$, which is what roundoff caused to happen.

How do we know that, if computed exactly, $x_n \to 6$ as $n \to +\infty$? Iterates x_n are functions of n expressed in a simple way as follows: Substitute $x_n := q_{n+1}/q_n$ into Muller’s recurrence to get a *Linear Homogeneous* recurrence $q_{n+2} = 111 \cdot q_{n+1} - 1130 \cdot q_n + 3000 \cdot q_{n-1}$. Well-known methods (see roots z above) provide its general solution $q_n = \alpha \cdot 100^n + \beta \cdot 6^n + \gamma \cdot 5^n$ for arbitrary constants α, β and γ . When these are chosen to match given initial values $x_0 := 11/2$ and $x_1 := 61/11$, say $\alpha = 0 \neq \beta = \gamma$, we find that, when computed exactly,

$$x_N = (6^{N+1} + 5^{N+1})/(6^N + 5^N) = 6 - 1/(1 + (1.2)^N) \to 6 \text{ as } N \to +\infty .$$

But roundoff alters (at first slightly) values computed from Muller’s recurrence to very near $X_N := (100^{N+1} \cdot t + 6^{N+1} + 5^{N+1})/(100^N \cdot t + 6^N + 5^N)$ with a nonzero tiny constant t as big as a rounding error. This $X_N \to 100$ as $N \to +\infty$. Were x_N computed from the recurrence $x_N := 11 - 30/x_{N-1}$ instead, roundoff would have little effect; do you see why?

Without the foregoing analyses, how would anyone discover that 100 is the wrong result when so many different arithmetics agree upon it? Should different arithmetics’ divergent intermediate results arouse suspicions? Alas, many everyday computations, including linear equation solving and eigensystems, often exhibit similar digressions among intermediate results while final results agree and are quite correct. In general there is no easy way to detect a wrongly computed result.

Problem 2: Shifted Legendre Polynomials

These polynomials appear in a few problems of Mathematical Physics concerned with functions on a line segment or a hemisphere; they also provide nodes x_j and weights w_j for *Gaussian Quadrature* formulas that approximate $\int_o^1 f(x)\,dx$ closely by a weighted average $\sum_j w_j f(x_j)$. There are many ways to define these polynomials. One is explicit; it is a

Rodrigues Formula: $P_N(x) := (d/dx)^N (x^2 - x)^N / N!$.

Another way starts with $P_{-1}(x) := 0$ and $P_0(x) := 1$ and obtains in turn $P_1(x)$, $P_2(x)$, $P_3(x)$, ..., $P_N(x)$, … from a

Recurrence Formula $P_N(x) := (2 - 1/N)(2x - 1)\cdot P_{N-1}(x) - (1 - 1/N)\cdot P_{N-2}(X)$.

Can you confirm that both definitions of $P_N(x)$ yield the same polynomial for every nonnegative integer N ? Can you confirm that $P_N(1\text{-}x) = (-1)^N P_N(x)$? Amazingly, all N+1 coefficients of $P_N(x)$ are integers; can you explain why?

Compute all the zeros of $P_{12}(x)$ and $P_{24}(x)$ to at least 12 sig. dec., and accompany your results with an account of how they were obtained and why you think they are as accurate as you claim.

Solution 2:

The Rodrigues formula yields $P_n(x) = (-1)^n\cdot\sum_{k\ge0} (-x)^k\cdot(n+k)\mathbf{!}/((k\mathbf{!})^2\cdot(n-k)\mathbf{!})$ subject to the understanding that $(n-k)\mathbf{!} = \infty$ when $k > n$. Substituting this formula into the given recurrence satisfies it after considerable labor. Substituting $x := y + 1/2$ into the Rodrigues formula yields $P_N(y + 1/2) = (d/dy)^N (y^2 - 1/4)^N / N\mathbf{!} = (-1)^N\cdot P_N(-y + 1/2)$, which confirms the allegation that $P_N(1\text{-}x) = (-1)^N P_N(x)$. Therefore the zeros of $P_N(x)$ are situated symmetrically about $x = 1/2$.

The coefficients c(j) of $P_{12}(x) = \sum_{1\le k\le13} c(j)\cdot x^{13-j}$ are all integers (why?) that fit into eight decimal digits and can therefore be computed exactly with ease in several ways to get …

k :	1	2	3	4	5	6	7	8	9	10	11	12	13
c(k) :	2704156	-16224936	42678636	-64664600	62355150	-39907296	17153136	-4900896	900900	-100100	6006	-156	1

Here is the MATLAB program `slegc.m` that computed this row **c** of coefficients:

```
function  c = slegc(n)
%  c = slegc(n)  produces a row of coefficients from which
%  polyval(c, x)  computes the  Shifted Legendre  polynomial
%   P[n](x) = c(1)*x^n + c(2)*x^(n-1) + ... + c(n+1)
%          := (d/dx)^n ((x-1)x)^n / n! = ((2n-1)(2x+1)P[n-1] - (n-1)P[n-2])/n
%  whose  c(n+1) = (-1)^n  and  0 <  all  P's  zeros  < 1 .
%  See also  slegt.m  for those zeros.  INEXACT if  n > 22 .
if  n < 1 ,  c = 1 ;  return,  end
c = zeros(n+1,2) ;   j = [2, 1] ;
c(1:2, :) = [ 1, 2 ;  0, -1 ] ;
for  k = 2:n
  j = 3-j ;   % ...  c(:,j(1))  belongs to  P[k-2],  c(:,j(2))  to  P[k-1] .
  d = [ 4*k-2; 1-2*k; 1-k ] ;
  c(1:k+1,j(1))= ([c(1:k+1,j(2)), [0;c(1:k,j(2))], [0;0;c(1:k-1,j(1))]]*d)/k ;
  end
c =  round(c(:,j(1))') ;  % ... a row of integers.
```

One way to compute the zeros of P_{12} uses MATLAB's `roots(c)`. Another way uses MATLAB's `fzero('pleg', guess, [], c)` with `pleg(x, c) = polyval(c, x)` and using each zero from `roots(c)` in turn as the `guess`. Alas, though MATLAB carries 53 sig. bits, about as precise as 15 - 16 sig. dec., the two sets of zeros disagree in digits as early as the tenth sig. dec.

Alleged Zeros of the Shifted Legendre $P_{12}(x)$

	`roots( sleg(12) )`	`fzero( 'pleg', …)`
1	0.9907803173102080	0.9907803171520745
2	0.9520586276890755	0.9520586282713346
3	0.8849513376839413	0.8849513371017566
4	0.7936589767059028	0.7936589771193403
5	0.6839157497188167	0.6839157495080903
6	0.5626167041824337	0.5626167042508767
7	0.4373832957592386	0.4373832957439742
8	0.3160842504994446	0.3160842505008112
9	0.2063410228566696	0.2063410228566984
10	0.1150486629028637	0.1150486629028469
11	0.04794137181476173	0.04794137181476259
12	0.009219682876640380	0.009219682876640380

Computed by Matlab 5.2 on a 68040-based Mac Quadra 950

Which, if any, of these zeros should we trust? Perhaps none, since the sums of pairs of zeros symmetrically situated about $x = 1/2$ should agree with 1 but actually disagree after the ninth or tenth digit after the decimal point in the first column, after the tenth or eleventh in the second. Should the second column be *Presumed Innocent until Proved Guilty* in the sense that it is no less accurate than its sums? Are ten or eleven sig. dec. adequate when twelve were requested?

While we mull over those questions, let's use our programs to do unto $P_{24}(x)$ what we did to $P_{12}(x)$. All 25 coefficients in row `c = slegc(24)` are integers again, but some are so huge they must have been rounded off. And then `z = roots(c)` cannot be right because its largest element $z(1) = 1.006815226068646 > 1$ whereas Rolle's Theorem and the Rodrigues formula tell us that every zero of $P_{24}(x)$ lies strictly between 0 and 1 . Worse, several of the alleged zeros z are complex with imaginary parts of the order of 0.02 . And after coaxing 24 distinct real zeros from `fzero('pleg', …, c)` and sorting them, we find that their sums of pairs agree with 1 at best to six digits, at worst to two. Eight to fourteen of the arithmetic's digits got lost.

Where do so many of the arithmetic's decimal digits go?

They get lost *after* the coefficient-row **c** is computed *even if it is exactly right*. Computing the zeros of P_n from those coefficients engenders rounding errors whose effect is about as bad as if each coefficient had been perturbed by as many as n units in its last digit carried (53rd sig. bit) by the arithmetic. This assertion is complicated to prove for `roots(slegc(n))` , simple to prove for `fzero('pleg', …, c)` which evaluates $P_n(x)$ repeatedly from an expression like

$$P_n(x) = (\ldots((c_1 \cdot x + c_2) \cdot x + c_3) \cdot x + \ldots + c_n) \cdot x + c_{n+1} \ .$$

The expression you see is not the expression you get. Each arithmetic operation, multiply or add, if it cannot be performed exactly, must be rounded off to the arithmetic's precision (here 53 sig.

bits). This is tantamount to multiplying the exact result by a factor like $(1 + \mu)$, about which we can know only that $|\mu| \le 1/2^{53}$, before storing the product as the computed result. Consequently we get not $P_n(x)$ but something like the long expression

$$(\dots((c_1 \cdot x \cdot (1+\mu_1) + c_2) \cdot x \cdot (1+\mu_2)^2 + c_3) \cdot x \cdot (1+\mu_3)^2 + \dots + c_n) \cdot x \cdot (1+\mu_n)^2 + c_{n+1} \, .$$

The last addition $\dots + c_{n+1}$ suffers no rounding since it is actually a subtraction that mostly cancels when x is near a zero of P_n .

Each factor like $(1+\mu)$ can be “moved” left in that long expression and attached to previous c_j’s , replacing c_1 by something like $c_1 \cdot (1+\bar{\mu}_1)^{2n-1}$, c_2 by $c_2 \cdot (1+\bar{\mu}_2)^{2n-2}$, c_3 by $c_3 \cdot (1+\bar{\mu}_3)^{2n-4}$, … . In effect, the value allegedly computed for $P_n(x)$ is actually *exactly* the value at x of one of infinitely many unknown polynomials whose coefficients differ from P_n ’s only in end-digits.

How do such perturbations in the last sig. digits of the coefficients of P_n affect its zeros?

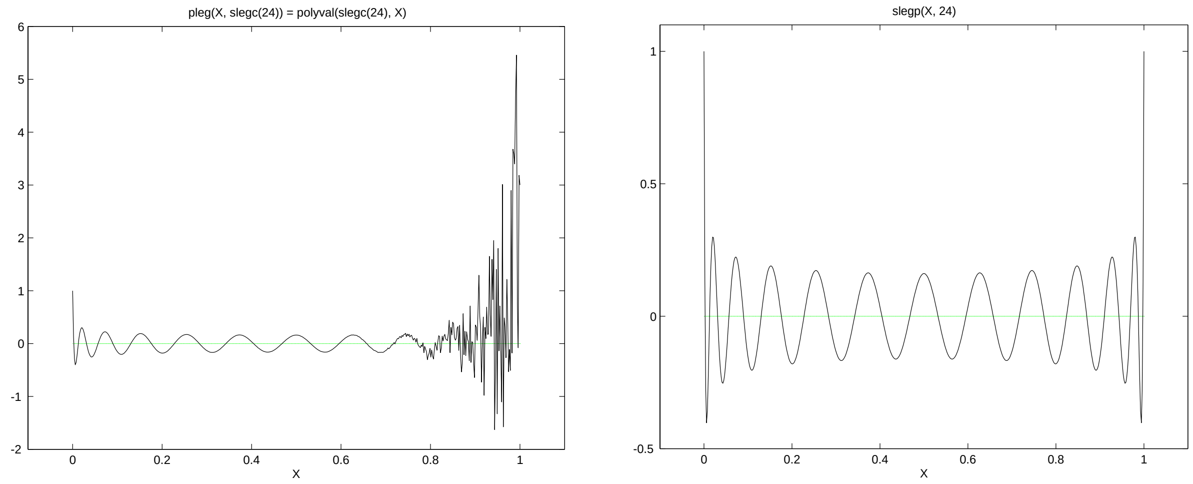
This question can be answered easily by experiment, by intentionally perturbing the coefficient row **c** by a little more than enough to swamp the effects of roundoff. After computing row **c** let `dc = [n:–1:0].*abs(c)*1e-14` and then compare `roots(c)` with `roots(c+dc)` and `roots(c–dc)` . The effect of perturbations in the 13th and 14th sig. dec. of the coefficients of P_{12} is to alter its larger zeros in their 6th sig. dec., smallest zero in its 14th. The larger zeros of P_{24} are altered in their first sig. dec. Evidently the larger zeros of P_n are “Ill-Conditioned” functions of its coefficients, the more so as the larger coefficients increase with increasing n . The larger zeros’ ill-condition arises from their *relatively* close clustering though they are just as closely clustered *absolutely* as are the smaller zeros. A better way to say this is that relatively tiny end-figure perturbations of the coefficients suffice to cause larger but not smaller perturbed zeros to coalesce, but that’s a story for another day.

To compute the larger zeros well (without just exploiting their symmetry about 1/2 and thus abandoning the use of that relationship to check on their accuracy) we *Eliminate the Middleman*: we compute the zeros of P_n from its definition, not its coefficients. We can get P_n ’s zeros by using recurrence directly to compute $P_n(x)$ = `slegp(x, n)` from a MATLAB program like this:

```
function  y = slegp(x, n)
%  slegp(x, n) = (d/dx)^n ((x–1)x)^n / n! = P[n](x)  is the  Shifted
%  Legendre Polynomial  of degree  n  computed from its recurrence
%  P[k+1](x) = ( (2k+1)(2x-1)P[k](x) - kP[k-1](x) )/(k+1)  starting
%  with  P[-1] = 0  and  P[0] = 1 .  Argument  x  may be a scalar or a
%  column vector  x = x(:) .  Only an integer scalar  n > -2  works.
if ((n~=round(n)) | (n<0)),  N = n ,
  error('slegp(x, N)  requires an integer  N > -2 .'), end
x = x(:) ;  L = length(x) ;  u = ones(L,1) ;
P = [u, u*0] ;  % ... = [ P[0],  P[-1] ]
if  (n<1),  y = P(:, 1-n) ;  return,  end
x21 = 2*x - 1 ;
for  k = 1:n,
     v = [2*k-1 ;  1-k]/k ;
     Q = P(:,1) ;  P(:,1) = Q.*x21 ;
     P = [ P*v, Q] ;  % ... = [ P[k](x), P[k-1](x) ]
   end
y = P(:, 1) ;
```

How much better than `pleg(x, slegc(n)) = polyval(slegc(n), x)` is this `slegp(x, n)` ? It's unobvious when $n = 12$, but when $n = 24$ some idea is conveyed by the following graphs:

polyval(slegc(24), x) slegp(x, 24)



Note the graphs' different vertical scales.

Evidently `polyval(slegc(n), x)` is less accurate than `slegp(x, n)` for larger values of `x` . Less obvious is that `polyval(slegc(n), x)` is the more accurate when `x` is tiny enough. The absolute (in)accuracy of `slegp(x, n)` is spread fairly uniformly over the interval $0 < x < 1$ because the recurrence preserves the identity $P_n(1–x) = (–1)^n \cdot P_n(x)$ exactly including roundoff whenever `x` is so chosen that $1–x$ can be computed exactly in binary floating-point, and such is certainly the case throughout $1/2 \le x \le 1$. (Can you see why? Note that $2x–1 = x – (1–x)$.)

The preservation of valuable relationships is as important to computation as to any other human activity.

The larger zeros of P_n can now be computed more accurately than before from `slegp(x, n)` by invoking MATLAB's `fzero('slegp', guess, [], n)` with n suitable guesses. These can be obtained as before from `roots(slegc(n))` if n is not too big. (12 isn't but 24 is.). Or they can be estimated from observed zero-crossings of plots of $P_n(x)$ against x , though the graphs above show why this scheme runs a risk that some of closely spaced zeros may be overlooked or estimated twice. Risks worsen as n gets bigger, and they can be exacerbated by the idiosyncratic way MATLAB's `fzero` carries out its search for a zero when supplied a single `guess` instead of an *Interval* `guess = [lower, higher]` that straddles the desired zero.

MATLAB's `fzero` works well enough when it is supplied with intervals that straddle desired zeros $x = z_k$ of $P_n(x)$, as do the known inequalities $\cos^2\big(k\pi/(2n+1)\big) < z_k < \cos^2\big((k–1/2)\pi/(2n+1)\big)$ for $k = 1, 2, 3, …, n$. However, these might not be known to everyone. On the other hand, `fzero` behaves erratically when its single `guess` is a known-to-be-close *Asymptotic* estimate

$$x_k := \cos^2\big((k–1/4)\pi/(2n+1) + \cot\big((k–1/4)\pi/(2n+1)\big)/(8n^2) \big) \approx z_k + O(1/n^3) .$$

These might not be known to everyone. Worse, `z(k) = fzero('slegpd', x(k), [], n)` finds some zeros z_k twice and overlooks others when n is too big (as is $n = 24$) despite that x_k is much closer to the desired zero than are adjacent zeros. MATLAB's `fzero` hastily casts too wide a net when it begins its search for a zero z_k near x_k , but that is a story for another day.

How can someone, who does not already know near enough where the desired zeros are, find *every one* of them *just once* apiece?

Deflation is a scheme designed to eliminate the risk of overlooking some zeros and computing others twice; it removes zeros already found. *Numerical* Deflation "cancels out" zeros already found so that they will not likely interfere with the search for other zeros: If zeros z_1, z_2, and z_3 have been computed, the zeros of $P_n(x)/\big((x–z_1)(z–z_2)(x–z_3)\big)$ are the remaining $n–3$ zeros z_k of $P_n(x)$. Submitting that quotient instead of $P_n(x)$ to a root-finder, say to MATLAB's `fzero`, continues the search for zeros with diminished sensitivity to the initial guess. Alas, numerical deflation has its failure modes. One is that roundoff's raggedness may lead to two approximations to one zero and none to another nearby. Less likely is that the search may alight accidentally upon a zero computed previously, hit 0/0 , and stop. Still, numerical deflation is worth trying.

Here is a MATLAB program `slegpd` to compute $P_n(x)/\big((x–z_1)(z–z_2)(x–z_3)(\ldots)(x–z_{j–1})\big)$ after `slegp` has computed $P_n(x)$:

```
function  y = slegpd(x,n, z,j)
%  slegpd(x,n, Z,j) = slegp(x, n)/prod(x-Z(1:j-1))
%  vectorized so that  plot(X, slegpd(X, n, Z, j))
%  will work assuming  X = X(:)  and  Z = Z(:) .
x = x(:) ;  j = round(j) ;  % trims  j  to integer.
if  j < 2,  y = slegp(x, n) ;  return,  end
Ux = ones(length(x), 1) ;
z = z(:) ;  zj = z(1:j-1) ;  Uz = ones(j-1, 1) ;
y = slegp(x, n)./prod( x(:,Uz)' - zj(:,Ux) )' ;
```

Still, of the four MATLAB couplets

```
z = zeros(n,1) ;
  for  j = 1:n,  z(j) = fzero('slegpd', 1, [], n, z,j) ;  end
. . . . . . . . . . . . . . . . . . . . . . . . _  Old versions of MATLAB need [],[] instead of [] .
z = zeros(n,1) ;
  for  j = 1:n,  z(j) = fzero('slegpd', 0, [], n, z,j) ;  end
. . . . . . . . . . . . . . . . . . . . . . . . _
z = zeros(n,1) ;  dg = 1e-8 ;  g = 1 - dg ;
  for  j = 1:n,  z(j) = fzero('slegpd', g, [], n, z,j) ;  g = z(j) - dg ;  end
. . . . . . . . . . . . . . . . . . . . . _______ . . . . . . . . . . . _______
z = zeros(n,1) ;  dg = 1e-8 ;  g = 0 + dg ;
  for  j = 1:n,  z(j) = fzero('slegpd', g, [], n, z,j) ;  g = z(j) + dg ;  end
```

all but the last malfunctioned for $n = 24$. The first two find only 8 of the zeros of P_{12} , only 14 or 16 of the zeros of P_{24} and they are found out of order. The third finds only 22 of the zeros of P_{24} , and out of order. The fourth finds all desired zeros as accurately as desired and in order. The same results were obtained from MATLAB 5.2 on a Mac Quadra 950 and on an IBM PC.

What should be done by someone who couldn't get `fzero` to work? Some other root-finding method like Newton's iteration $x \longrightarrow x – P_n(x)/P_n'(x)$ can be tried. The derivative $P_n'(x)$ can be computed in any of several ways. One way uses the row `c = slegc(n)` of coefficients of

$P_n(x)$ to compute a row `d = [n:-1:1].*c(1:n)` of coefficients of $P_n'(x) =$ `polyval(d, x)`. Another way computes $P_n'(x)$ simultaneously with $P_n(x)$ from Horner's recurrence used to compute $P_n(x) = (\dots((c_1 \cdot x + c_2) \cdot x + c_3) \cdot x + \dots + c_n) \cdot x + c_{n+1}$. The simultaneous recurrences look like this in MATLAB :

```
        p = c(1) ;  p1 = 0 ;
        for  n = 2:n+1,
                p1 = p1.*x + p ;     % ... the derivative of ...
                p = p.*x + c(n) ;    % ... Horner's recurrence.
            end  % ...  yielding  p = Pn(x)  and  p1 = Pn'(x) .
```

However, schemes based upon the coefficient-row **c** inherit its defect: Even if **c** is computed exactly right, its huge coefficients amplify roundoff intolerably as it propagates unless n or x is relatively small. As we have seen above (but not proved yet), no such defect afflicts the three-term recurrence for $P_n(x)$; it can be augmented to compute $P_n'(x)$ simultaneously thus:

```
    $P_{-1}(x) := P_{-1}'(x) := P_0'(x) := 0$ ;   $P_0(x) := 1$ ;
    for  $N = 1, 2, 3, \dots, n$  in turn do
        $P_N'(x) := \big( (2N-1)\cdot\big((2x-1)\cdot P_{N-1}'(x) + 2P_{N-1}(x)\big) - (N-1)\cdot P_{N-2}'(x) \big)/N$ ,
        $P_N(x) := \big( (2N-1)(2x-1)\cdot P_{N-1}(x) - (N-1)\cdot P_{N-2}(x) \big)/N$ .
```

Can you see how to augment `slegp.m` above to deliver $P_n'(x)$ as well as $P_n(x)$? How should you modify the expression $x - P_n(x)/P_n'(x)$ to deflate previously computed zeros z_1, z_2, and z_3 numerically without a new program to compute the derivative of $P_n(x)/\big((x-z_1)(z-z_2)(x-z_3) \big)$? Hint: $P'/P = \log(P)'$.

Another formula for the derivative is $P_n'(x) = n\cdot\big((2x-1)\cdot P_n(x) - P_{n-1}(x) \big)/\big(2x(x-1)\big)$. It looks too simple. Can you prove it by induction? Can you see why it should be avoided, especially when x is near 0 or 1 ?

With or without the derivative, with or without deflation, the zeros of P_n computed by finding where `slegp(x, n)` vanishes or reverses sign seem accurate enough. They pass two tests. One compares 1 against sums of pairs of zeros situated symmetrically about 1/2 ; all sums agree with 1 in all but its last (53rd) bit or two. Another test compares $|c(n+1)| = 1$ with the product of all n computed zeros and coefficient c(1) ; they agree in all but the last few bits. Two such tests prove nothing unless one fails, in which case some (we know not which) computed zero(s) would be proved inaccurate.

Can convincing evidence be found for the accuracy of the computed zeros of `slegp(x, n)` ? Yes; the desired zeros of $P_n(x)$ can be computed in an utterly different way that corroborates strongly the accuracy now in question and can be proved to suffer from roundoff about the same as do zeros computed from the three-term recurrence, though their rounding errors are different.

Set the infinite column $\mathbf{p}(x) := [P_0(x), P_1(x), P_2(x), P_3(x), P_4(x), \dots, P_{n-1}(x), P_n(x), \dots]'$; set the infinite diagonal matrix $\mathbf{M} := \mathrm{Diag}([1, 3, 5, 7, 9, \dots, 2n-1, 2n+1, \dots])$; and set the infinite tridiagonal matrix

$$\mathbf{Y} := \left[\begin{array}{cccccccccc}
0 & 1 & & & & & & & & \\
1 & 0 & 2 & & & & & & & \\
 & 2 & 0 & 3 & & & & & & \\
 & & 3 & 0 & 4 & & & & & \\
 & & & 4 & 0 & \dots & & & & \\
 & & & & \dots & \dots & n-1 & & & \\
 & & & & & n-1 & 0 & n & & \\
 & & & & & & n & 0 & n+1 & \\
 & & & & & & & n+1 & \dots & \dots \\
 & & & & & & & & \dots & \dots
\end{array}\right] .$$

In each instance the rows bracketed between pairs of "…" are rows #n and #n+1 . Now the three-term recurrence for Shifted Legendre Polynomials can be written as a matrix equation $\mathbf{Y}\cdot\mathbf{p}(x) = (2x–1)\cdot\mathbf{M}\cdot\mathbf{p}(x)$. This equation simplifies a little after the substitution $\mathbf{q}(x) := \sqrt{\mathbf{M}}\cdot\mathbf{p}(x)$, or $Q_n(x) := \sqrt{2n–1}\cdot P_n(x)$, to become $\mathbf{T}\cdot\mathbf{q}(x) = (2x–1)\cdot\mathbf{q}(x)$ in which $\mathbf{T}$ is an infinite symmetric tridiagonal matrix with $[0, \dots, 0, (n–1)/\sqrt{(4(n–1)^2–1)}, 0, n/\sqrt{(4n^2–1)}, 0, 0, \dots]$ in its row #n .

Obtain $\mathbf{T}_n$ and $\mathbf{q}_n(x)$ by deleting all rows and columns after #n from $\mathbf{T}$ and $\mathbf{q}(x)$ respectively. If z is any zero of $P_n(x)$ then $\mathbf{T}_n\cdot\mathbf{q}_n(z) = (2z–1)\cdot\mathbf{q}_n(z)$ because $Q_n(z) = P_n(z) = 0$. Conversely if $\mathbf{T}_n\cdot\mathbf{q}_n(z) = (2z–1)\cdot\mathbf{q}_n(z)$ then $P_n(z) = Q_n(z) = 0$. In other words, all zeros z of $P_n(x)$ can be obtained from eigenvalues $t = 2z–1$ of $\mathbf{T}_n$ without ever computing the values or coefficients of P_n ; in fact $z = (1+t)/2$. Here is a MATLAB program `slegt.m` that computes $\mathbf{T}_n$:

```
function  Tn = slegt(n)
%  slegt(n)  is the symmetric  n-by-n  tridiagonal matrix whose
%  eigenvalues  t ,  all between  -1  and  1 ,  are the zeros of
%  the  nth  Legendre Polynomial;  z = (1+t)/2  yields the zeros
%  of the shifted  Legendre  polynomial  Pn(x) = slegp(x, n) ,  q.v.
n = round(n) ;  if n < 2 ,  Tn = zeros(n) ;  return,  end
x = [1:n-1] ;  x2 = x+x ;
x = x./sqrt((x2+1).*(x2-1)) ;
Tn = diag(x,1) + diag(x,-1) ;
```

Computed eigenvalues t of $\mathbf{T}_n$ can be tested for accuracy in two ways. First, they should come in pairs with opposite signs since $\mathbf{T}_n$ is *Similar* to $–\mathbf{T}_n = \mathbf{D}_n^{–1}\cdot\mathbf{T}_n\cdot\mathbf{D}_n$ via a diagonal matrix $\mathbf{D}_n = \text{Diag}[+1, –1, +1, –1, \dots]$. Second, reversing the rows and columns of $\mathbf{T}_n$ should not change its eigenvalues, but will change them slightly because of different rounding errors. We find that `t = sort(eig(slegt(n)))` is very nearly sign-symmetric since `t + flipud(t)` has elements all of the order of $10^{–15}$. Likewise `t – sort(eig(flipud(fliplr(slegt(n)))))` . These tests do not take the effects of roundoff in the elements of $\mathbf{T}_n$ into account. To test their effects, compare 1 with `c(1)*prod(z)` where `z = (1 ± t)*0.5` are the desired zeros of P_n . This test of the computed column **z** of zeros produced a difference of the order of $10^{–14}$, so their accuracy seems to be corroborated. It can be proved, but that is a story for another day.

Tabulated on the next page are zeros z computed using a Pentium's 10-byte floating-point:

Zeros of Shifted Legendre Polynomials $P_n(x)$				
n = 12		n = 24		
0.990780317123359625	1	0.997593609998510680	0.467971553568697187	13
0.952058628185237428	2	0.987364277985654749	0.404440566263191845	14
0.884951337097152344	3	0.969137276001366379	0.342478660151918313	15
0.793658977143308724	4	0.943207763502200517	0.283103246186977431	16
0.683915749499090097	5	0.910000992986951461	0.227289264305580232	16
0.562616704255734458	6	0.870062095789277182	0.175953174031512215	18
0.437383295744265542	7	0.824046825968487785	0.129937904210722818	19
0.316084250500909903	8	0.772710735694419768	0.0899990070130485390	20
0.206341022856691276	9	0.716896753813022569	0.0567922364977994829	21
0.115048662902847656	10	0.657521339848081687	0.0308627239986336208	22
0.0479413718147625710	11	0.595559433736808155	0.0126357220143452509	23
0.00921968287664037467	12	0.532028446431302813	0.00240639000148931993	24

Tabulated values are in error in only their last digit displayed.

Appendix: Brute-force verification that Rodrigues' formula for $P_n(x)$ satisfies its recurrence:

Differentiate n times the *Binomial Expansion* of $(x^2 - x)^n/n!$ to get

$$P_n(x) = (-1)^n\cdot\sum_{0\le k\le n}(-x)^k\cdot(n+k)!/\left((k!)^2\cdot(n-k)!\right)$$
$$= (-1)^n\cdot\sum_{0\le k\le n} X_k\cdot(n+k)!/(n-k)! \quad \text{where } X_k := (-x)^k/(k!)^2 .$$

Every coefficient $(n+k)!/\left((k!)^2\cdot(n-k)!\right) = {}^{n+k}C_{2k}\cdot{}^{2k}C_k$ is the product of two binomial coefficients and therefore an integer. Note that $x\cdot X_k = -(k+1)^2\cdot X_{k+1}$. Substitute this into P_n's three-term recurrence formula in the form

$$R_N = (-1)^{N+1}\cdot\left((N+1)\cdot P_{N+1} - (2N+1)(2x-1)\cdot P_N + N\cdot P_{N-1} \right)$$
$$= \sum_{0\le k\le N+1} X_k\cdot(N+1)\cdot(N+1+k)!/(N+1-k)! + \sum_{0\le k\le N} X_k\cdot(2N+1)(2x-1)\cdot(N+k)!/(N-k)!$$
$$+ \sum_{0\le k\le N-1} X_k\cdot N\cdot(N-1+k)!/(N-1-k)!$$

$$= \sum_{0\le k\le N+1} X_k\cdot(N+1)\cdot(N+1+k)!/(N+1-k)! - \sum_{0\le k\le N} 2X_{k+1}\cdot(k+1)^2\cdot(2N+1)\cdot(N+k)!/(N-k)!$$
$$- \sum_{0\le k\le N} X_k\cdot(2N+1)\cdot(N+k)!/(N-k)! + \sum_{0\le k\le N-1} X_k\cdot N\cdot(N-1+k)!/(N-1-k)!$$

$$= \sum_{0\le k\le N+1} X_k\cdot(N+1)\cdot(N+1+k)!/(N+1-k)! - \sum_{0\le k\le N+1} 2X_k\cdot k^2\cdot(2N+1)\cdot(N+k-1)!/(N-k+1)!$$
$$- \sum_{0\le k\le N} X_k\cdot(2N+1)\cdot(N+k)!/(N-k)! + \sum_{0\le k\le N-1} X_k\cdot N\cdot(N-1+k)!/(N-1-k)!$$

I wish I knew how to persuade Mathematica or Maple to perform the foregoing and following manipulations:

Coeff. of X_{N+1} : $(N+1)\cdot(2N+2)! - 2(N+1)^2\cdot(2N+1)! = 0$.

Coeff. of X_N : $(N+1)\cdot(2N+1)! - 2\cdot N^2\cdot(2N+1)\cdot(2N-1)! - (2N+1)\cdot(2N)! = 0$.

Coeff. of X_k for $0 \le k \le N-1$: $(N+1)\cdot(N+1+k)!/(N+1-k)! - 2\cdot k^2\cdot(2N+1)\cdot(N+k-1)!/(N-k+1)!$

$$- (2N+1)\cdot(N+k)!/(N-k)! + N\cdot(N-1+k)!/(N-1-k)!$$
$$= \left((N+k-1)!/(N-k+1)!\right)\cdot\left((N+1)(N+k)(N+k+1) - 2\cdot k^2\cdot(2N+1) - (2N+1)(N+k)(N-k+1) + N(N-k)(N-k+1) \right) = 0 .$$

Therefore $R_N = 0$; the Rodrigues formula does satisfy the three-term recurrence.

Our *Shifted Legendre Polynomials* $P_n(x)$ are called “$P_n^*(x)$” by Urs W. Hochstrasser, who uses “$P_n(x)$” for (unshifted) Legendre Polynomials in ch. 22, as does Irene A. Stegun in ch. 8 of her *Handbook of Mathematical Functions …* published in 1964 by the then National Bureau of Standards (now NIST) and subsequently reprinted by Dover, N.Y. Their $P_n(z) = P_n^*((z+1)/2) = (d/dz)^n(z^2–1)^n/(2^n n!)$, so our $P_n(x)$ is his $P_n^*(x) = P_n(2x–1)$. The asymptotic estimates and inequalities for the zeros of P_n come from his #22.16.6 . The formula that should not be used to compute the derivative comes from her #8.5.4 .

Problem 3: Fibonacci Numbers

When the Fibonacci numbers F_n are generated by the recurrence

$$F_0 = 0\ ,\quad F_1 = 1\ ,\quad \text{and}\quad F_{n+1} = F_n + F_{n-1}\quad \text{for}\ \ n = 1, 2, 3, \dots\ \text{in turn,}$$

computing just F_N for a given large N takes time at least proportional to N . For example

$$F_{1471} = 11785114478791471849880\dots15229 \qquad (\text{ 308 decimal digits}).$$

If all we need is F_N to, say, 12 sig. dec., how can we compute it much faster than that, and accurate to within a few units in the last digit carried? How do you know your chosen method is that accurate?

Solution 3:

Here is a MATLAB program `fib.m` to compute quickly F_n = round-to-nearest-integer($\mu^n/\sqrt{5}$) wherein $\mu := (1 + \sqrt{5})/2$. The program has to compensate for rounding the value of μ to an 8-byte floating-point number u . Otherwise u^n could differ from μ^n by an amount that grows with n , which can be as big as 1474 before u^n overflows. The compensation is devious; the program exploits massive subtractive cancellation, which inherits error but adds none new, to estimate rounding error $v := \mu – u \approx –5.432/10^{17}$ to about 5 sig. dec. Then $\mu^n \approx u^n + n\cdot(v/u)\cdot u^n$ after terms of order $(n\cdot v/u)^2\cdot u^n$ are ignored.

```
function  f = fib(n)
%  fib(n) = F(n) = the  nth  Fibonnacci  number for  n >= 0
%         = F(n-1) + F(n-2)  starting from  F(1) = F(2) = 1
%  computed very quickly without computing earlier  F's .
%  Overflow turns  fib(n)  into  NaN  for  n > 1474 .
%  Fib(n)  accepts arrays  n  of non-consecutive integers.
%                                    (c)  W. Kahan  2004
n = round(n) ;  if any(n < 0)
        N = n
        error('  fib(N)  accepts only  N >= 0 .'),  end
s = sqrt(5) ;  u = (1+s)*0.5 ;  %  = (1+sqrt(5))/2 - v
%  v = ( 207/128 - u ) + 31/(8192*s + 18304) ;  %  ...  3 more digits UNUSED
v = ( 13255/8192 - u ) - 1201/(33554432*s + 75030528) ;  % ... 5 more digits
un = u.^n ;
f = round( ( un + ((v/u)*n).*un )/s ) ;
```

How accurate is `fib.m` ? How much more accurate is it than if compensation v were omitted? And how do we know? For MATLAB versions 5.x and 6.x , the relative error in `fib.m` turns out to be tiny enough to be best measured in *Ulp*s — *U*nits in the *L*ast *P*lace stored for floating-point variables in question.

We define ulp(x) to be the gap between the two floating-point numbers nearest x . If x is a floating-point number then ulp(x) is the difference between x and its nearest neighbor. For MATLAB's 8-byte floating-point carrying 53 sig. bits, ulp(x) = `eps` = $1/2^{52} \approx 2.22/10^{16}$ when $1 < x \le 2$. Computing ulp(x) can be tricky because of variations in the way different computers round off floating-point arithmetic and handle exponent over/underflow. For example MATLAB's ulp(0.0) should be `0.5^1074` $\approx 4.94/10^{324}$ but can be `realmin = 0.5^1022` $\approx 2.225/10^{308}$ on a few aberrant computers. For the purposes of this problem about Fibonacci numbers,

```
      min( abs( (x + (0.51*eps)*x) – x ),  abs( (x – (0.51*eps)*x) – x ) )
```

computes ulp(x) well enough on PCs, Macs and SPARCs. If real number x is approximated by floating-point X then its error is (X–x)/ulp(x) ulps.

MATLAB versions 5.x and 6.x produce values `fib(n)` differing from F_n by from –2.5 ulps to +1.5 ulps; about half of that error is generated by MATLAB's computation of u^n , so it's pretty good. Were the compensating term $v/u \approx -3.3572/10^{17}$ omitted, values of `fib(n)` would come out too high by about from 0.15·n ulps to 0.3·n ulps for n > 75 . How do we know?

Here is a MATLAb program `fibs.m` that computes Fibonacci numbers F_n correctly rounded. They are all integers, but rounded off to fit into 8-byte floating-point numbers for n > 78 .

```
function  [F, dF] = fibs(n)
%  F = fibs(n) = [F1, F2, F3, ..., Fn]  is a row of
%  Fibonacci  numbers computed slowly but accurately
%  from their recurrence  F(n+1) = F(n) +  F(n-1)
%  started at  F(1) = F(2) = 1 .  If  n > 78  roundoff
%  is attenuated by  Compensated Summation,  and then
%  [F, dF] = fibs(n)  returns also a compensating term
%  dF  such that  F + dF  would be more accurate if its
%  sum did not just round off to correctly rounded  F .
%  Overflow  turns  Fn  into  NaN  when  n > 1476 .
%                               (c) W. Kahan  2004
F = ones(1, n) ;  dF = zeros(1, n) ;
if  (n < 3),  return,  end
for  k = 3:n
    s = ((dF(k-2) + dF(k-1)) + F(k-2)) + F(k-1) ;
    t = (((F(k-1) - s) + F(k-2)) + dF(k-1)) + dF(k-2) ;
    F(k) = s + t ;             % ...  rounded off,  and
    dF(k) = (s - F(k)) + t ;   % ...  is the rounding error.
  end
```

Though `f = fib(1:n)` runs rather faster than `[F, dF] = fibs(n)` , the error in the latter's `F` is smaller than 0.5 ulps and approximated well enough by dF that the error in the former's `f` can be computed fairly well as `((f-F) - dF)./ulp(F)` ulps. If compensated summation were not used in `fibs(n)` , but instead it computed simply `F(k) = F(k-1)+F(k-2)` , its error for n > 78 would look like a random walk ranging between about –13 ulps and +3 ulps.

The last dozen finite Fibonacci numbers F delivered by `[F, dF] = fibs(1476)` are tabulated below together with the leading several digits of their corrections dF .

n	F(n)	dF(n)
1465	6.56761920344308045**E**305	-2.924700176543**E**288
1466	1.06266310963374150**E**306	-4.236201169795**E**289
1467	1.71942502997804946**E**306	3.267580021670**E**289
1468	2.78208813961179081**E**306	1.462388127012**E**290
1469	4.50151316958984058**E**306	-1.329354354469**E**290
1470	7.28360130920163108**E**306	3.251534256190**E**290
1471	1.17851144787914723**E**307	-4.314821065575**E**290
1472	1.90687157879931034**E**307	-1.063286809385**E**290
1473	3.08538302667845756**E**307	-5.378107874961**E**290
1474	4.99225460547776815**E**307	-3.138939855353**E**291
1475	8.07763763215622521**E**307	1.312850130988**E**291
1476	1.30698922376339934**E**308	-1.826089724365**E**291

If you think you can copy, align and add the decimal digits of F(n) and dF(n) to get the leading 29 sig. dec. of F_n , you have forgotten that *What You See is **Not** What You Get* from binary floating-point arithmetic. 18 sig. dec. of F(n) displayed here more than suffice for MATLAB's command `load nFdF -ascii` to reproduce exactly the 8-byte binary floating-point number F(n) from which the displayed string of digits was obtained by binary-to-decimal conversion. At least 40 of the 53 leading sig. bits of dF(n) will be reproduced too, but then adding it to F(n) will be futile because MATLAB will round the sum back to F(n) . Can you see how to tease the leading 25 (say) sig. dec. of F_n from MATLAB without using its Symbolic Toolbox? The process is tricky. Here, obtained by using the Symbolic Toolbox, is a value to compare with yours:

$$\begin{aligned} F_{1476} = {} & 1306989223763399318036311553802719830983924439074126407260066594601927930704792 \\ & 31740288681087770177210954631549790122762343222469369396471853667063684893626 \\ & 608441474499413484628009227558189696347433489829164249540627441359698656154072 \\ & 76492410653721774590669544801490837649161732095972658064630033793347171632. \end{aligned}$$

For an earlier treatment of Problem 3 on earlier versions of MATLAB see
http://www.cs.berkeley.edu/~wkahan/~MxMuleps.pdf .

What's worth remembering about the foregoing three problems? They illustrate some of the hazards of approximate computation, and how to circumvent them.

First, roundoff gets amplified intolerably by a computation only when its data comes too close to a singularity at which certain derivatives become infinite or nonexistent. In Problem 1, x_{34} and x_{340} are too close to x_∞ which is a discontinuous function of initial values x_0 and x_1 . When n is big in Problem 2 the row **c** includes some coefficients so huge that, even computed exactly, they amplify intolerably later rounding errors incurred during the computation of $P_n(x)$ for x near 1 . In Problem 3 the difference between $\mu^n \approx (u + v)^n$ and u^n becomes intolerable slowly as n becomes huge though v is just one rounding error. To diagnose numerical misbehavior,

Seek the Singularity.

Some singularities seem intrinsic to the computational problem we wish to solve, as is the jump discontinuity of x_∞ in Problem 1 . Then we attach pejoratives like "ill-conditioned" ot "ill-posed" to the problem, as if to blame it for the troubles we must overcome to solve it. Some singularities are accidents of the method chosen to solve the problem, as are the huge coefficients **c** we need not but could compute to find the zeros of P_n . Then we attach the pejorative term "numerically unstable" to the chosen method, as if to blame it for our computation's inaccuracy, though the method may work well on other problems for all we know. More than disparaging a method or a problem, what matters is that we understand the source of numerical misbehavior well enough to detect and diagnose it accurately, and then remove or evade or learn to live with it. That's not so easy as it sounds.

What is the true purpose of Problem 1 ? If it is to compute the result of repeated applications of Muller's Recurrence to uncertain data x_0 and x_1 , then the result's near-discontinuity at this data is an attribute of the result too important to leave unmentioned and too unobvious for casual computation to expose. On the other hand, if the purpose is to compute a long sequence of terms x_n as if x_0 and x_1 were exact, then Muller's Recurrence is a numerically unstable way to do that; far more accurate is the simple but unobvious recurrence $x_n := 11 – 30/x_{n–1}$. Elementary though Problem 1 seems at first, it cannot be solved satisfactorily without mathematical analysis that transcends what any computer program can be expected nowadays to do automatically.

Problem 2 exposes the curse of high dimensionality and/or high degree. If `roots(slegc(n))` were tested only at $n < 12$ its gross inaccuracy at larger n would go unnoticed until too late. Only scrutiny of a graph like the ones on p. 7 or the coefficients in `c = slegc(n)` reveals how badly their huge magnitudes degrade the computation of the larger zeros of $P_n(x)$ as n grows.

Degradation like that, which usually blights some zeros of characteristic polynomials of matrices of other than small dimensions, astonished us in the 1950s when we tried to apply methods that had often worked in previous decades to bigger problems on new electronic computers. Now the (usually) prudent policy for floating-point arithmetic is to compute eigenvalues directly from a matrix rather than from the zeros of its computed characteristic polynomial *even if its coefficients are computed exactly.* (Some people still think rounding errors in the coefficients' last digits are solely to blame for all the trouble, but $P_{23}(x)$ provides a counter-example.) The homogeneous linear three-term recurrence satisfied by polynomials $P_n(x)$ connects them to a tridiagonal matrix $\mathbf{T}_n$ = `slegt(n)` whose eigenvalues computed by MATLAB's `eig(Tn)` are, as usual, much more reliable than the zeros of $P_n(x)$ computed from its coefficients `c = slegc(n)` by MATLAB's `roots(c)` or `fzero(…)` and `polyval(c, x)` using floating-point of just the same precision.

The weasel-word "usually" is necessary because some zeros, like the smallest few of $P_n(x)$, may be computable faster and more accurately from its coefficients than from the eigenvalues of its associated matrix. The same may be true for all the zeros of *Lacunary* polynomials most of whose coefficients vanish, and for the relevant one or two zeros of a polynomial arising from financial calculation of an interest rate or *Internal Rate of Return* on investment.

Problem 3 provides the simplest illustration that over/underflow thresholds, not precision, are what limit the accuracy of exclusively floating-point computations since arbitrarily high precision can be simulated, albeit at some cost in complexity and time. If built properly into programming languages, higher precisions would simplify computation and enhance its reliability enormously.