An Assignment for Math. 128 B due Thurs. 6 Apr. 2006

1. What is the largest integer N for which MATLAB's C = poly(1:N) produces the row of coefficients of $p(x) := \prod_{1\le k\le N} (x - k) = C(1)\cdot x^N + C(2)\cdot x^{N-1} + \ldots + C(N)\cdot x + C(N+1)$ exactly despite roundoff, and what evidence have you to support your claim?

2. Having computed N and C in problem 1, let G = [0, abs(C(2:N+1))] and find the smallest |ß| such that C + ß*G is the row of coefficients of a polynomial — MATLAB would compute it as polyval(C+ß*G, x) — with a real double zero, and explain why your ß is fairly accurate.

Model Solutions:

(Covering far more methods than any one student is expected to try.)

1. The largest N = 18 . Four ways come to mind to determine whether MATLAB has computed coefficient row `C = poly(1:N)` uncorrupted by rounding errors:

•(o) Use the *Inexact Flag* mandated by IEEE Standard 754 for Floating-Point Arithmetic. This flag is accessible through some ***C*** compilers, but not through *Java* nor MATLAB, alas.

•(i) Compare MATLAB's `C` with the coefficients computed exactly by an automated algebra system like *Maple*, *Mathematica* or *Derive*. Roundoff during Binary - Decimal conversion could obscure MATLAB's `C` for all we know, so its elements should be displayed in *Hexadecimal* using MATLAB's format `hex` and compared with the exact values, all integers, displayed in hexadecimal adjusted for floating-point normalization. When N = 19 the coefficients of x^5 and x^4 cannot fit exactly into 53 sig. bits.

•(ii) Compute `Z = polyval(C, [1:N]')` . It should be a column `Z` of N zeros if no rounding error has corrupted `C = poly(1:N)` nor `Z = polyval(C, [1:N]')` . But what if `Z` has been corrupted? This question undermines faith in `polyval` , so we replace it below by a MATLAB program `rpolyval` intended to *Iteratively Refine* `polyval`'s accuracy.

•(iii) Each step of MATLAB's recurrence that computes `C = poly(1:N)` uses only one subtract and one multiply, so augmenting each step by the computation of a residual may expose any rounding error that corrupts the step. MATLAB program `cpoly` below tries that.

MATLAB's Binary Floating-Point Arithmetic:

Normally its variables are stored in 8-byte words with 53 sig. bits conforming to the *Double Precision* format of IEEE Standard 754. A description of the standard convenient for students in this course is posted at <http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>. Normally MATLAB rounds its every atomic arithmetic operation (+, –, *, /, √‾) on real variables to 53 sig. bits. MATLAB does this on most UNIX machines like Sun SPARCs and SGI MIPs.

Most versions of MATLAB can do something else when multiplying matrices on a few other very widely used computers, so different versions of MATLAB on the same hardware, and the same version on different hardware, can deliver different results for many computations.

IBM's *Power PC* architecture, used also in Apple *Power Mac*s, *iMac*s, *G3*s and *G4*s, has a *Fused Multiply-Add* operation that evaluates expressions like ±x ± y*z with only one rounding error after the add/subtract operation; the product x*y is generated exactly in a double-width internal register. MATLAB 5.2 on those Apple computers uses fused multiply-add operations only for the accumulations of scalar products $\sum_k$ X(i,k)*Y(k,j) during matrix multiplications X*Y provided the matrices fit into the computer's cache. Consequently the MATLAB expression `[eps–1, 1]*[eps+1; 1]` evaluates to 0 but `[1, eps–1]*[1; eps+1]` evaluates to eps^2 on those computers, and they evaluate `[pi, -pi]*[pi; pi]` to 1.429…e-16 instead of 0 . This behavior will sometimes help certain polynomial expressions to be evaluated more accurately than if each multiply were rounded to 53 sig. bits before the subsequent rounded add/subtract.

Some computer architectures can perform atomic floating-point operations in a few extra-precise registers carrying 64 sig. bits instead of just 53. Among these processors are the near-ubiquitous Intel x86, *Pentium*, *Centrino*, … and their clones by AMD in computers running Microsoft *Windows* or, less numerous, *LINUX*. Similar extra-precise registers exist also in the current Intel *Itanium*, and in the old Motorola 68040 in my old Apple *Macintosh Quadra*, and in a few other processors now mostly forgotten. Access to the extra-precise registers is denied by many programming languages (like *Java*) and by most compilers (like Microsoft's), which is why some versions of MATLAB can accumulate scalar products extra-precisely during at least some matrix multiplications on those processors, and other versions cannot. Among versions that can are versions 3.5, 4.2 and 5.2 on old Quadras, and versions 3.5, 4.2, 6.5 and maybe 7.0 on Wintel PCs; version 6.5 requires the command `system_dependent('setprecision', 64)` to enable extra-precise accumulation during multiplications of matrices small enough to fit into the computer's cache. When extra-precise accumulation is in effect,

```
        t = 0.5^32 ;   u = [t-1, 1]*[t+1; 1] ,  v = [1, t-1]*[1; t+1]
```

delivers not 0 but $1/2^{64}$ = 5.421…e-20 for *both* `u` and `v` . But after that MATLAB 7.0 does something weird; see `weird.m` below. What it may be doing is adding scalar products like $\sum_k$ X(i,k)*Y(k,j) in a different order than for k = 1, 2, 3., … in turn. For instance, two sums $\sum_k$ X(i,2k)*Y(2k,j) and $\sum_k$ X(i,2k–1)*Y(2k–1,j) could be accumulated in separate registers in parallel and then added, thus achieving higher speed overall by keeping the arithmetic pipeline full. Whatever the order of summation, extra-precise accumulation will help certain polynomial expressions to be evaluated more accurately than if each multiply were rounded to 53 sig. bits before the subsequent rounded add/subtract.

Checking MATLAB's poly(1:N) against cpoly(N) :

Given a column or row $Z = [z_1, z_2, z_3, …, z_N]$ we expect `C = poly(Z)` to deliver the row C of coefficients C(k) of the polynomial

$$p(x) := \texttt{prod(x-Z)} = \prod_{1\le k\le N} (x–z_k) = \sum_{0\le k\le N} C(k+1)\cdot x^{N–k} = \texttt{polyval(C, x)}$$

in the absence of roundoff. The recurrence that generates the coefficients works as follows: Let E be the row of coefficients of polynomial $q(x) := \prod_{1\le k<N} (x–z_k) = p(x)/(x–z_N)$. Then

```
                    C = [1, -Z(N)]*[[E, 0]; [0, E]] ;
```

in other words $C(k) = E(k) – z_N\cdot E(k–1)$ for k = 1, 2, 3, …, N+1 assuming E(0) := E(N+1) := 0 .

Note that, when every $z > 0$ (as is the case for `poly(1:N)`), the coefficients in E and C alternate in sign and so every step of the recurrence generates integer coefficients each bigger in magnitude than its ancestors. Therefore no roundoff can occur until some magnitude exceeds 2^{53} and perhaps not even then if that magnitude is divisible by a sufficiently big power of 2 . Because every coefficient of `poly(1:17)` is smaller in magnitude than 2^{51} it fits easily into 53 sig. bits, so the biggest N for which `poly(1:N)` is exactly right must be at least 17 .

As N increases through N = 17, 18, 19, … the first rounding error to blight the recurrence can occur in one of only three places: The multiply $z_N \cdot E(k–1)$, the subtract $E(k) – z_N \cdot E(k–1)$, or the store C(k) = … if the previous two operations have been performed extra-precisely. The first assignment's first task amounts to detecting that first rounding error when N > 17 .

If that first rounding error occurs at the multiply operation and is not obscured by a subsequent rounding error at the subtract or store operations, a different first rounding error will very likely occur if the array Z is permuted before `poly(Z)` is invoked, so we might expect it to differ from, say, `poly(flipud(Z(:)))` , which reverses the order of the zeros in Z without changing the polynomial p(x) nor its coefficients C except for roundoff. N = 20 was the first integer at which `norm(poly(1:N) - poly(N:-1:1), inf)` turned out nonzero; it did so on all my Macs and PCs, and for all versions of MATLAB mentioned so far. Therefore `poly(1:20)` and/or `poly(20:-1:1)` is inexact; we cannot yet say which. Only an unlikely cancellation of rounding errors could leave one of them exactly right, so we may well suspect that the biggest N for which `poly(1:N)` must be exact cannot exceed 19 .

To determine that biggest N more reliably, we must try to detect the recurrence's first rounding error by computing a residual that exploits exact cancellation when sufficiently nearby numbers are subtracted. After storing `C = [1, -Z(N)]*[[E, 0]; [0, E]]` approximately, we compute the difference `dC = [1, -1, -Z(N)]*[[E, 0]; C; [0, E]]` in which only the multiplication by Z(N) can generate a rounding error, and this won't happen if Z(N) is 1 or 2 . Therefore we reverse the order of the zeros of p(x) by setting `Z = N:-1:1` so that the first rounding error will arise after either the subtract or the store operation. This error will be exposed when `dC` is not all zeros. The MATLAB program `cpoly(N)` below does the trick:

`cpoly(N)` agrees with `poly(1:N)` for N = 1, 2, 3, …, 17, 18 and 19 but signals inexactness first when N = 19 at the recurrence's last step where Z(19) = 1 . The first rounding error occurs at the subtract or store operation and is captured perfectly. `cpoly(N)` disagrees with `poly(1:N)` for $N \ge 20$ and signals inexactness at the recurrence's last three or more steps. Consequently …

N = 18 seems to be the biggest N for which `poly(1:N)` is exact.

Comparing MATLAB's poly(1:N) with Recomputations in Redirected Rounding Modes :
This somewhat flakey scheme works only for a few versions of MATLAB on one or two almost ubiquitous computer architectures.

Nowadays practically all desktop and laptop computer hardware conforms to *IEEE Standard 754 for Binary Floating-Point Arithmetic*. Almost no compilers support this Standard properly. Consequently certain capabilities of the hardware are accessible to almost no programmers. One such capability is *Directed Rounding*: By *default* (in the absence of a command to do otherwise) every floating-point operation, if not exact like 2.0 + 2.0 = 4.0 , *Rounds to Nearest*, which means that the rounding error cannot exceed half a unit in the last bit retained. The Standard mandates that a programmer be able to command the hardware to round in her choice of any of three other directions: *Round Up* towards $+\infty$, *Round Down* towards $-\infty$, and *Round Towards Zero*. The rounding error in these *Directed Rounding Modes* is strictly smaller than one unit in the last bit retained but may exceed half a unit.

Moreover the drafters of the Standard intended (but we failed to make this clear enough) that the programmer be able to rerun any subprogram in each of those four *Rounding Modes* so as to generate four samples of the influence of redirected roundoff upon the results. Rounding modes' effects upon a Math. library of transcendental functions like log(…) and cos(…) are uncertain because the Standard does not specify them.

The rounding mode could be altered in MATLAB 3.5 on PCs and Macintoshes only via an intervention by the computer's operating system. MATLABs 4.x and 5.x thwarted every such intervention. MATLABs 6.5 and 7.0 on PCs offer a way to alter rounding modes to some extent:

```
system_dependent('setround', DIRECTION)
```

with	`DIRECTION = 0.5`	for Round to Nearest,
	`DIRECTION = 0.0`	for Round Towards Zero,
	`DIRECTION = +inf`	for Round Up towards $+\infty$,
	`DIRECTION = -inf`	for Round Down towards $-\infty$.

Ideally these commands should redirect the roundings of every subsequent rational operation (+, –, *, /), the algebraic operation $\sqrt{\ }$, Binary ↔ Decimal Conversion, and some conversions to integers. Instead these commands redirect the rational operations' roundoff but leave sqrt(…) and round(…) unaltered. (In MATLAB 3.5 redirection altered all of them.)

To test the commands' effects, interleave them among repeated evaluations of an expression like

```
[ x+y;  x-y;  x*y;  x/y;  sqrt(x);  round(y);  log(x);  str2num(DECIMAL) ]
```

in which the variables have suitable values preassigned *outside* the MATLAB script or function containing repetitions of this expression. Preassignment "*outside*" prevents MATLAB's script-and-function compiler from evaluating what are deemed to be constant expressions in the default rounding mode at compile-time, rather than later in the altered modes at run-time. Differences between different evaluations of that long expression reveal the effects of redirected roundings.

Differences appear among the four evaluations of `poly(1:N)` under all four rounding modes only when $N > 18$. This test proves that `poly(1:18)` is uncontaminated by roundoff. The test does not rule out the possibility that, under the default mode, rounding errors in `poly(1:N)` may cancel out for some $N \ge 19$, but that seems unlikely.

Checking MATLAB's mod(poly(1:N), 2^m) against polymod2(m, 1:N) :

Given a column or row $Z = [z_1, z_2, z_3, \dots, z_N]$ we expect `C = poly(Z)` to deliver the row C of coefficients C(k) of the polynomial

$$p(x) := \texttt{prod(x-Z)} = \prod_{1\le k\le N} (x-z_k) = \sum_{0\le k\le N} C(k+1)\cdot x^{N-k} = \texttt{polyval(C, x)}$$

in the absence of roundoff which may blight the last few bits of some coefficients. When all the zeros in Z are integers, so are all the coefficients in C , and their last few bits can be computed perfectly by *Modular Arithmetic*: MATLAB's `mod(K, M) = K - floor(K/M)*M` is intended to deliver a remainder between 0 and M when integer K is divided by integer M . Versions of MATLAB earlier than 5.x must use `rem(K, M)` instead. Alas, roundoff can interfere when K gets too big. For instance `mod(2.0.^[52:67], 3)` and `rem(2.0.^[52:67], 3)` produce …

[1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2] for v's. 3.5 & 4.2 on a Mac Quadra, and v. 4.2 on a PC.

[1 2 1 2 1 2 1 2 1 2 1 2 1 0 0 0] for v. 3.5 on a PC.

[1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0] from mod for v. 5.2 on a Mac Quadra, and from both mod and rem for v. 5.2 on a Power Mac, and for v's. 5.3, 6.5 & 7.0 on a PC.

[1 2 1 2 4 8 16 32 64 … 8192] from rem for v. 5.2 on a Mac Quadra, and from both mod and rem for v's. 6.5 &7.0 after `system_dependent('setprecision', 64)` .

Only the first is correct.

We avoid all such roundoff-induced aberrations entirely by restricting $M := 2^m$ for some positive integer $m < 27$. This is what the MATLAB program `polymod2(m, L)` supplied below does. It computes exactly the last m bits of the integer coefficient array C for a polynomial whose zeros are all integers in an array L . Therefore `any(polymod2(m, 1:N) == mod(C, 2^m))` should be 0 whenever `C = poly(1:N)` has been computed exactly, and 1 otherwise, provided m is not too small. With $m := 26$ this exactness test is passed when $1 \le N \le 18$ and failed when $N \ge 19$ for all versions of MATLAB and on all computers mentioned above. This test *proves* that …

$N = 18$ is the biggest N for which `poly(1:N)` is exact.

Checking MATLAB's C = poly(1:N) against polyval(C, 1:N) :

Given a column or row $Z = [z_1, z_2, z_3, \ldots, z_N]$ we expect `C = poly(Z)` to deliver the row C of coefficients C(k) of the polynomial

$$p(x) := \texttt{prod(x-Z)} = \prod_{1\le k\le N} (x - z_k) = \sum_{0\le k\le N} C(k+1)\cdot x^{N-k} = \texttt{polyval(C, x)}$$

in the absence of roundoff. Then we expect `P = polyval(C, Z)` to generate a vector of zeros. But, for all we know now, rounding errors inside `polyval` may generate misleading results:

- Some elements of `P` may be nonzero even though `C` is exactly right.
- All elements of `P` may be zero even though `C` is slightly inexact.

When the expression `any(polyval(poly(1:N), 1:N))` is evaluated for N = 1, 2, 3, … in turn it produces 0 for $1 \le N \le 18$ and 1 for $N \ge 19$, and does so on PCs and Macs for every version of MATLAB mentioned above. On any one machine these results imply that N = 18 is the biggest N for which `poly(1:N)` is exact provided no rounding error occurs inside `polyval`; but this proviso is problematical. It seems unlikely for at least some of the larger values of N .

Because binary multiplication is exact, rounding errors inside `polyval` can be restricted to add/subtract operations by evaluating `any(polyval(poly(1:N), [1,2,4,8,16]))` just for $N \ge 16$. These evaluations produce 0 for $16 \le N \le 18$ and 1 for $N \ge 19$ again, leaving the same problematical question about roundoff inside polyval unanswered.

Since the value of a polynomial evaluated by *Horner's Recurrence* (*i.e.* nested multiplications) can be construed as the last element of the solution of a system of linear equations, its accuracy may be improved by iterative refinement provided a residual can be computed extra-precisely. The MATLAB program `rpolyval(C, X)` below does so whenever MATLAB accumulates scalar products extra-precisely, and does so sometimes when they are accumulated on Power Macs using fused multiply-adds. The residual is computed simultaneously with the refinement's correction `dp` in the statement `dp = [C(m), p, op, dp]*x2` .

When the expression `any(rpolyval(poly(1:N), [1,2,4,8,16]))` is evaluated just for N = 16, 17, 18, … in turn it produces 0 for $16 \le N \le 18$ and 1 for $N \ge 19$, and does so on PCs and Macs for every version of MATLAB mentioned above. On any one machine these results suggest that N = 18 is the biggest N for which `poly(1:N)` is exact provided rounding errors inside `rpolyval` have been offset adequately by iterative refinement.

What follows is an attempt to assess how much more accurate than `polyval` is `rpolyval` . The assessments vary with the version of MATLAB and the computer on which it runs. Each test compares `polyval` and `rpolyval` with a sufficiently accurate evaluation of the polynomial under test. Here are a few test results:

Test #1: Polynomial `p(x) = prod(x - [1:18])` has coefficients `C = poly([1:18])` exactly.

With extra-precisely accumulated scalar products

x	prod(x–[1:18])	rpolyval(C, x)	polyval(C, x)
3.875	-76286600135.58403	-76286600135.58403	-762866001*56*.
15.125	-76286600135.58403	-76286600135.58403	-76*164317290*.
$4 + 1/2^{10}$	510102608.499438*6*	510102608.4*639658*	51010*4157*.
$15 – 1/2^{10}$	510102608.499438*5*	510*073677.3643785*	*615021841*.

With fused multiply-adds

x	prod(x–[1:18])	rpolyval(C, x)	polyval(C, x)
3.875	-76286600135.58403	-76286600*218.6133*	-762866001*56*.
15.125	-76286600135.58403	-763*99896318.5432*	-76*164317290*.
$4 + 1/2^{10}$	510102608.49943*9*	510103*060.829811*	51010*4157*.
$15 – 1/2^{10}$	510102608.49943*8*	5*20948386.105919*	*615021841*.

With no extra-precise arithmetic

x	prod(x–[1:18])	rpolyval(C, x)	polyval(C, x)
3.875	-76286600135.58403	-762865*99734.74647*	-762866001*56*.
15.125	-76286600135.58403	-76*164317290*.	-76*164317290*.
$4 + 1/2^{10}$	510102608.499438*7*	510103*342.7073206*	51010*4157*.
$15 – 1/2^{10}$	510102608.499438*4*	5*29906517.7198682*	*615021841*.

MATLAB 5.3 on a PC produces results like those in the last tabulation except that its second-last column duplicates the last because iterative refinement by `rpolyval(C, X)` is thwarted by the way this version of MATLAB accumulates scalar products: It accumulates $\sum_k$ X(i,k)*Y(k,j) with no extra-precise arithmetic and in descending order k = …, 3, 2, 1 instead of the ascending order v. 5.2 on Macs and v. 6.5 on PCs follow when matrices fit in the computer's cache. Two evaluations `[-1, 1, eps]*[1; 1; eps]` and `[eps, 1, -1]*[eps; 1; 1]` expose the order.

MATLAB 7.0 on a PC accumulates scalar products extra-precisely (sometimes differently before `system_dependent('setprecision', 64)` than after), and in an order that depends upon the dimensions of matrices in `weird` ways I have not yet figured out. Here are its test results:

MATLAB 7.0 on a PC

x	prod(x–[1:18])	rpolyval(C, x)	polyval(C, x)
3.875	-76286600135.584	-76286600136.*1108*	-762866001*56*.
15.125	-76286600135.584	-76286*635021*.*3008*	-76*164317290*.
$4 + 1/2^{10}$	510102608.49943*9*	510102607.*926758*	510104*157*.
$15 – 1/2^{10}$	510102608.49943*8*	510*040748*.*883789*	*615021841*.

Test #2: Roundoff corrupts `C = poly(1:19)` producing the row of coefficients of a polynomial p(x) := `prod(x – [1:19]) + 16*(x-2)*x*x*x*x`. Roundoff corrupts `polyval(C, x)` so that it differs from p(x). Iterative refinement usually brings `rpolyval(C, x)` nearer p(x) provided residuals are accumulated extra-precisely. Then, for instance, `for x = [-4:23]` we find every `rpolyval(C, x) == p(x)` but half the values `polyval(C, x) ~= p(x)`. And the differences `rpolyval(C, x) - p(x)` are a few orders of magnitude smaller than `polyval(C, x) - p(x)` for `x = 0.5 + [-1:22]`, and for `x = [sqrt(2); exp(1); pi; sqrt(377)]`. Activate extra-precise accumulation by MATLAB 6.5 via `system_dependent('setprecision', 64)`. This command is unnecessary for v. 7.0 on PCs but does alter a few results slightly. Without extra-precise accumulation (v. 5.3 on PCs and v. 6.5 without that command), and regardless of fused multiply-adds (v. 5.2 on Power-Macs), `rpolyval` is no more accurate than `polyval` for this test.

These test results accord with what we had learned elsewhere about the iterative refinement of computed solutions of linear systems. With extra-precisely accumulated residuals, they and the solution's accuracy are improved by refinement. It may worsen accuracy without extra-precise accumulation if the linear system is ill-conditioned.

The Significance of Exactly Computed Coefficients:

Why should we care whether `C = poly(1:N)` is computed exactly? A reason will appear when the accuracy of `R = roots(C)` is assessed for N = 1, 2, 3, … . Rounding errors will cause R to depart from a permutation of the integers [1, 2, 3, …, N]. Whose rounding errors? Blaming `roots` for rounding errors committed by `poly`, or *vice-versa*, would be unfair, and would spoil attempts to compute ß for part 2 of this assignment. Tabulated below are some computed sets of `roots(poly(1:18))` from different versions of MATLAB run on a few different computers.

Versions of MATLAB’s Computed `roots(poly(1:18))`

v. 3.5 on PC & Mac Quadra	v. 4.2 on PC	v. 4.2 on Mac Quadra	v. 5.2 on Mac Quadra	v. 5.2 on Power Mac
17.99999	17.99999	18.000006	18.00001	18.000009
17.0001	17.00006	16.99995	16.9999	16.99992
15.9996	15.9998	16.0002	16.0005	16.0004
15.001	15.0005	14.9997	14.999	14.999
13.998	13.9994	14.0004	14.002	14.002
13.002	13.0006	12.9997	12.998	12.997
11.999	11.9997	12.0001	12.002	12.003
11.0008	11.0001	11.00006	10.999	10.998
9.9997	9.99998	9.99991	10.0004	10.001
9.00009	8.99999	9.00005	8.99991	8.9996
7.99998	8.000005	7.99998	8.000004	8.0001
7.000001	6.999999	7.000005	7.000005	6.99997
5.99999999	6.0000001	5.9999993	5.999999	6.000005
4.999999994	4.999999993	5.00000006	5.0000002	4.9999995
3.999999999	4.0000000006	3.9999999992	3.99999998	4.00000004
3.0000000002	2.99999999991	2.9999999998	3.0000000006	2.999999999
1.99999999999	2.000000000004	2.000000000007	1.999999999993	2.00000000002
1.0000000000001	0.99999999999997	0.99999999999995	1.00000000000002	1.00000000000007

v. 5.3 on PC	v. 6.5 on PC	v. 6.5 on PC (64)	v. 7.0 on PC	v. 7.0 on PC (64)
17.999996	18.000008	18.000009	17.999997	17.999997
17.00003	16.99991	16.99992	17.00001	17.00003
15.99991	16.0005	16.0004	16.00002	15.9999
15.0002	14.999	14.9990	14.9997	15.0004
13.9998	14.002	14.002	14.0008	13.9992
13.0003	12.997	12.998	12.998	13.001
11.9996	12.003	12.002	12.002	11.999
11.0004	10.998	10.999	10.998	11.0008
9.9997	10.0007	10.0006	10.001	9.9996
9.0001	8.9998	8.9998	8.9995	9.0002
7.99995	8.00002	8.00005	8.0002	7.99996
7.00001	7.000005	6.99999	6.99996	7.000008
5.999998	5.999998	6.000001	6.000006	5.9999991
5.0000002	5.0000003	4.9999999	4.9999995	5.00000006
3.999999996	3.99999998	4.00000001	4.00000003	3.9999999991
2.9999999997	3.00000000001	2.9999999997	2.9999999993	2.99999999997
2.00000000002	2.00000000002	2.000000000005	2.000000000001	2.0000000000006
0.9999999999998	0.9999999999998	0.99999999999996	1.00000000000007	0.99999999999997

“on PC (64)” means after the command `system_dependent('setprecision', 64)`

Why are accuracies so similar though results vary from one computer to another and from one version of MATLAB to another? Without knowing the details about QR iteration and matrix multiplication within MATLAB's `eig(…)`, invoked within `roots(C)` to compute its values as eigenvalues of the *Companion Matrix* of polynomial `polyval(C, x)` , we have to speculate:

• Results differ because rounding errors differ during the different orderings and precisions of the matrix multiplications within the QR-iterations that reduce the *Hessenberg Form* of the companion matrix to an upper-triangular *Schur* form with the desired eigenvalues (roots) on its diagonal. The different orderings are intended to take near-optimal advantage of the computer's memory architecture, especially its cache(s). Precisions differ according to whether extra-precise accumulation is enabled during matrix multiplication, or whether multiply-adds are fused.

• Though computed eigenvalues (roots) differ, their accuracies are similar because these depend heavily upon MATLAB's `eig` program's thresholds chosen to stop QR-iteration as soon as the Hessenberg matrices' sub-diagonal elements have become small enough to be deemed negligible.

So sensitive to those rounding errors and thresholds are some eigenvalues (roots) that computing them loses most of the digits carried by the arithmetic. Roots near 13 in the tabulations above are so senstive, and consequently are called "ill-conditioned". They are ill-conditioned no matter how we try to compute them from the coefficients `C = poly(1:18)` so long as our procedure commits rounding errors whose effect is tantamount to perturbing the coefficients' end-figures. Roundoff is like that in `roots(C)` and in `polyval(C, x)` , so computing a root `r` from either

```
 r = fzero( 'polyval(poly(1:18), x)', [12.5, 13.5], eps)
```
or
```
rr = fzero('rpolyval(poly(1:18), x)', [12.5, 13.5], eps)
```
(refined)

yields r = 13**.**00003 from an unrefined polyval, or
rr = 13**.**00003 in MATLAB 5.3 on a PC (refinement thwarted) ,
rr = 12**.**99994 in MATLAB 6.5 on a PC (refinement not extra-precise) ,
rr = 12**.**999999 in MATLAB 5.2 on a Power Mac (helped by its fused multiply-add),
rr = 13**.**0000001 in MATLAB 5.2 on a Mac Quadra and v. 6.5 on a PC (… 64),
rr = 12**.**9999998 in MATLAB 7.0 on a PC (performs extra-precise refinement),
instead of the correct root 13 .

Why are ill-conditioned roots so sensitive to tiny perturbations in a polynomial's coefficients? Their hypersensitivity is explained if comparably tiny perturbations can alter the multiplicities of ill-conditioned roots. This is what we shall see happen in part 2 of this assignment.

2. The smallest $|ß|$ belongs to $ß \approx 1.4217594218419e\text{-}14$ for which $p(x) + ß\cdot g(x)$ has a double zero at $x \approx 13.513354361516$. Here

$$p(x) := \prod_{1\le k\le N} (x-k) = C(1)\cdot x^N + C(2)\cdot x^{N-1} + \dots + C(N)\cdot x + C(N+1) \quad \text{and} \quad N = 18 ,$$

so coefficient row `C = poly(1:N)` exactly, and

$$g(x) := |C(2)|\cdot x^{N-1} + \dots + |C(N)|\cdot x + |C(N+1)| > 0 \text{ for all } x \ge 0 .$$

It is easy to confirm that $g(x) = (-1)^N\cdot(p(-x) - (-x)^N) = \prod_{1\le k\le N} (x+k) - x^N$. This implies soon that $|p(x)/g(x)| > 1/2$ if $x \le 0$, so $p(x) + ß\cdot g(x)$ cannot have a nonpositive double zero unless $|ß| > 1/2$. If $|ß|$ is to be tiny, the desired real double zero z must be positive .

Now, z is a double zero of $p(x) + ß\cdot g(x)$ just when both $p(z) + ß\cdot g(z) = 0$ and the derivative $p'(z) + ß\cdot g'(z) = 0$; and then $ß = -p(z)/g(z) = -p'(z)/g'(z)$, whence follows a polynomial equation $p'(z)\cdot g(z) - g'(z)\cdot p(z) = 0$. This is the equation that must be solved for some of its $2N-2 = 34$ roots z . Let's not compute the polynomial's coefficients and invoke `roots(...)` or `fzero(polyval(...))` ; they're likely to produce poor approximations from coefficients blurred by roundoff if recent experience is a guide. Instead the polynomial equation will be turned into an equivalent rational equation $p'(z)/p(z) - g'(z)/g(z) = 0$. In this equation $p'(x)/p(x)$ can be computed accurately enough from $p'(x)/p(x) = \sum_{1\le k\le N} 1/(x-k)$ because its graph is fairly steep in the range that will matter later: $(p'(x)/p(x))' = -\sum_{1\le k\le N} 1/(x-k)^2 < -8$ if $1 < x < N$.

(Can you see why "… < –8 " ? Actually it's "… < – 8.874… " , but the exact value won't matter to what follows.)

A more complicated expression provides an analogous (and unobvious) way to compute

$$g'(x)/g(x) = \Big(\prod_{1\le k\le N} (x+k)\cdot\sum_{1\le j\le N} 1/(x-j) - N\cdot x^{N-1}\Big)/\Big(\prod_{1\le k\le N} (x+k) - x^N\Big) = \dots$$
$$= N/x - (1/x)\cdot\Big(\sum_{1\le k\le N} k/(x+k)\Big)/\Big(1 - \prod_{1\le k\le N} x/(x+k)\Big) \quad \text{after some algebra.}$$

This expression leads to

$$f(x) := x\cdot\big(p'(x)/p(x) - g'(x)/g(x)\big) = \dots \text{ (more algebra)}$$
$$= \sum_{1\le k\le N} k/(x-k) + \Big(\sum_{1\le k\le N} k/(x+k)\Big)/\Big(1 - \prod_{1\le k\le N} x/(x+k)\Big) .$$

Now the rational equation $f(z) = 0$ has the same roots $z > 0$ as has the polynomial equation $p'(z)\cdot g(z) - g'(z)\cdot p(z) = 0$. To help locate those roots we examine the *Poles* of $f(x)$ (where it becomes infinite). For $K = 1, 2, \dots$ and N we find $f(x) \approx K/(x-K)$ when $x \approx K$, so $f(x)$ reverses sign as x crosses the pole at $x = K$. Therefore $f(x)$ also reverses sign at least once as x increases from one such pole to the next. We have found at least $N-1$ positive zeros of $f(x)$.

Are there any other positive zeros of $f(x)$? No; here is why:

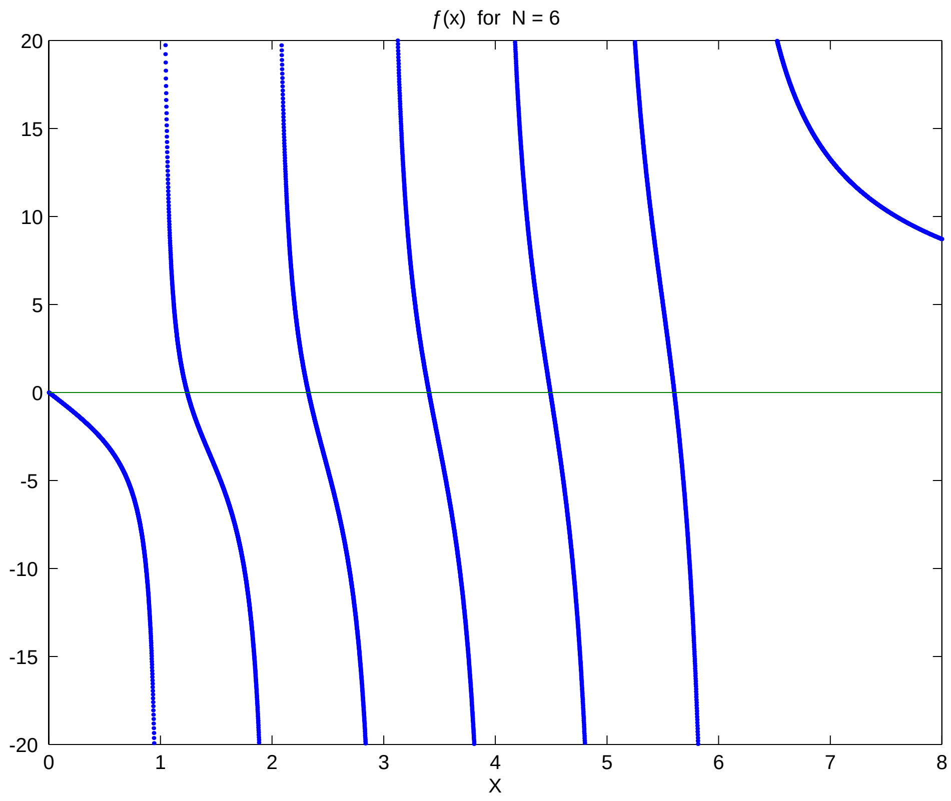
$f(x) < 0$ while $0 < x < 1$. This is true because then $g(x) > 0$, $g'(x) > 0$ and

$$f(x)/x = p'(x)/p(x) - g'(x)/g(x) < p'(x)/p(x) = \sum_{1\le k\le N} 1/(x-k) < 0 .$$

$f(x) > 0$ for all $x > N$. To see why this is true, verify that $(N-1)\cdot g(x) - x\cdot g'(x) > 0$ for all such x , so $p'(x)/p(x) - g'(x)/g(x) > \sum_{1\le k\le N} 1/(x-k) - (N-1)/x = 1/x + (1/x)\cdot\sum_{1\le k\le N} k/(x-k) > 0$.

Therefore all the positive zeros of $f(x)$ lie strictly between its poles at $x = 1, 2, 3, \dots$ and N .

How many zeros of $f(x)$ lie between adjacent poles? The easiest way to answer this question is to plot $f(x)$. It is plotted below for $N = 6$; the plot for $N = 18$ is similar but busier.



Apparently $f(x)$ vanishes just once between adjacent poles. This is confirmed by a proof (best skipped during a first reading) that, when $1 < x < N$, the derivative

$$f'(x) := \big(x\cdot p'(x)/p(x) - x\cdot g'(x)/g(x) \big)'$$

is negative. The first term $\big(x\cdot p'(x)/p(x) \big)' = \big(\sum_{1\le k\le N} x/(x-k) \big)' = -\sum_{1\le k\le N} k/(x-k)^2 < -8\cdot\sqrt{2}$.

To handle the second term $\big(-x\cdot g'(x)/g(x) \big)'$ we shall first determine a rough bound for every zero $\zeta = -\xi + \imath\eta$ of $g(x) = \prod_{1\le k\le N} (x + k) - x^N$. We find every $\mathrm{Real}(\zeta) = -\xi \le -1/2$ because otherwise, were $\xi < 1/2$, we would find that $|k-\xi + \imath\eta|^2 > |-\xi + \imath\eta|^2$ for all $k \ge 1$, and this would imply that $\prod_{1\le k\le N} |\zeta + k| > |\zeta|^N$ whence $g(\zeta) \ne 0$. Thus, $g(x) = -C(2)\cdot\prod_{1\le k\le N-1} (x - \zeta_k)$ in which every $\zeta_k = -\xi_k + \imath\eta_k$ has $\xi_k \ge 1/2$. Moreover, because $g(x)$ has real coefficients, its every non-real zero $\zeta = -\xi + \imath\eta$ comes paired with its complex conjugate zero $\bar{\zeta} = -\xi - \imath\eta$.

Now the second term $\big(-x\cdot g'(x)/g(x) \big)' = -\big(\sum_{1\le k\le N} x/(x-\zeta_k) \big)' = \sum_{1\le k\le N} \zeta_k/(x-\zeta_k)^2$ can be handled; this sum will be shown to be negative: In this sum every term $\zeta/(x-\zeta)^2$ with a real zero $\zeta = -\xi \le -1/2$ has the form $-\xi/(x+\xi)^2 < 0$. Every pair of complex conjugate terms adds up to $\zeta/(x-\zeta)^2 + \bar{\zeta}/(x-\bar{\zeta})^2 = \big(\zeta\cdot(x-\bar{\zeta})^2 + \bar{\zeta}\cdot(x-\zeta)^2 \big)/|x-\zeta|^4 = -2(\xi^2 + x\cdot\xi + \eta^2)/|x-\zeta|^4 < 0$. Therefore $f'(x) < -8\cdot\sqrt{2} \approx -11.3137$ for $1 < x < N$ and consequently $f(x)$ must vanish just once between adjacent poles just as appears from its plotted graph. (Actually $f'(x) < -15.475$ when $N = 18$.)

MATLAB program `betas(N)` appended below computes all $N-1$ positive zeros z of $f(x)$ and the corresponding values of ß from which the smallest is selected to complete the assignment.

MATLAB Programs Cited Above:

```
function  [m, n] = weird(k)
%  [m, n] = weird(k)  tests  Matlab's  scalar products of
%  vectors of length  2k .  If no extra-precise arithmetic is
%  used  [m, n]  should be  [0, 0] ,  as it is for  Matlabs
%  5.3 and 6.5  on  Wintel PCs.  If scalar products are
%  accumulated extra-precisely to  64  sig. bits before they
%  are delivered rounded to  53  sig. bits,  [m, n]  should
%  be  [k, k] ,  as it is for  Matlab 3.5 and 4.2  on  PCs
%  and old  680x0-based Macs,  and for  Matlab 6.5 on  PCs
%  after the command  system_dependent('setprecision', 64) .
%  In  Matlab 5.2  on  Power Macs  their fused multiply-adds
%  produce  [m, n] = [1, 0] .  But something weird and,  so
%  far,  unexplained happens in  Matlab 7.0  on  Wintel PCs:
%   k   m   n      k   m   n      k   m   n      k   m   n
%   1   1   1      5   5   3      9   9   5     13   9  17
%   2   2   1      6   8   8     10   8  16     14   8  16
%   3   3   2      7   9   9     11   9  17     15   9  17
%   4   4   2      8   8   4     12   8  16     16   8   8
%                                  W. Kahan,  29 March 2006
if  ((k<1)|(k~=round(k))),  k = k
  error('weird(k) requires a positive integer  k.'),  end
u = ones(1,k) ;
a = -2281422937 ;  b = 4042815511 ;  %...  a*b = 1 - 2^63
c = 2^31 ;  d = c+c ;  e = 1-d ;  f = 1+d ;
%  c = 2^31 ,  d = 2^32 ,  e = 1 - 2^32 ,  f = 1 + 2^32 .
CA = [c*u; a*u] ;  CA = CA(:) ;  %... = [c a c a c a ...]
DB = [d*u; b*u] ;  DB = DB(:) ;  %... = [d b d b d b ...]
m = CA'*DB ;  %... = (2^63 + (1 - 2^63))*k   if exact
ED = [e*u; d*u] ;  ED = ED(:) ;  %... = [e d e d e d ...]
FD = [f*u; d*u] ;  FD = FD(:) ;  %... = [f d f d f d ...]
n = ED'*FD ;  %... = ((1 - 2^64) + 2^64)*k   if exact

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

function  C = cpoly(n, m)
%  C = cpoly(n, m)  is the row of coefficients of a
%  polynomial whose zeros are  [m:n] ,  but differs
%  from  poly(m:n)  by checking whether  C  is free
%  from rounding errors,  as it is on a  Mac Quadra
%  950  for  (n,m) = (18,1), (19,2), (20,4), (21,6),
%  (22,7), (23,9), (24,10), (25,12), ... .
%  If  m  is omitted it defaults to  m = 1 .
if ( nargin < 2 ),  m = 1 ;  end
L = n-m+1 ;  C = zeros(1, L+1) ;  C(1) = 1 ;
for  k = (-n):(-m)
    E = C ;  oE = [0, E(1:L)] ;
    C = [1, k]*[E; oE] ;
    dC = [1, -1, k]*[E; C; oE] ; %...  residual
    if any(dC(:))  n_m_k = [n, m, -k] ,
        disp(' cpoly(n, m)  was computed inexactly.')
      end,  end
```

```
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

function  C = polymod2(m, L)
%  C = polymod2(m, L)  is the row of coefficients  mod 2^m
%  of a polynomial whose zeros are all integers in array  L .
%  This works for  0 < integer m < 27  in  Matlabs 5.2 - 7.0.
if ((m ~= round(m))|(m < 1)|(m > 26 )),  m = m
    error('polymod2(m,L) needs  0 < integer m < 27 .'),  end
M = 2^m ;  L = L(:) ;  if  any(L ~= round(L))
    error('polymod2(m,L) needs L to be all integers.'),  end
L = mod(L, M) ;  n = length(L) ;
C = zeros(1, n+1) ;  C(1) = 1 ;
for  k = 1:n
    oE = [0, C(1:n)] ;
    C = mod( [1, -L(k)]*[C; oE], M) ;  end

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

function  P = rpolyval(C, X)
%  P = rpolyval(C, X)  is an array of the same size as  X
%  containing the respective values of the polynomial
%  p(x) = C(1)*x^n + C(2)*x^(n-1) + ... C(n)*x + C(n+1)
%  wherein  n+1 = length(C) .  The computed values have
%  been  Iteratively Refined  once in a way that exploits
%  whatever extra-precise arithmetic or fused multiply-
%  add operation is available during matrix multiplication.
[rx,cx] = size(X) ;  X = X(:) ;  P = X ;  n = length(C) ;
for  k = 1:length(X)
    x1 = [1; X(k)] ;  x2 = [1; -1; X(k); X(k)] ;
    p = 0 ;  dp = 0 ;
    for  m = 1:n
        op = p ;
        p = [C(m), op]*x1 ;
        dp = [C(m), p, op, dp]*x2 ;  end %... m
    P(k) = p + dp ;  end %... k
P = reshape(P, rx,cx) ;

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

function  Y = f(X, n)
%  f(x, n) = sum([1:n]./(x-[1:n])) + sum([1:n]./(x+[1:n]))
%                                    ----------------------
%                                    1 - prod(x./(x+[1:n]))
%  for an array  x .  If omitted,  n  defaults to  18 .
if (nargin < 2),  n = 18 ;  end
[rx, cx] = size(X) ;  X = X(:) ;  L = length(X) ;
N = [1:n] ;  Y = X ;
for  j = 1:L
      x = X(j) ;
    y = sum(N./(x-N))+sum(N./(x+N))/(1 - prod(x./(x+N))) ;
    Y(j) = y ;  end
Y = reshape(Y, rx, cx) ;

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
function  [B, Z] = betas(n)
%  [B, Z] = betas(N)  provides columns of candidates for class
%  assignment #2 :  B = two columns of candidates for  beta
%  (they should match),  and  Z = a column of corresponding
%  double zeros of  p(x) + beta*g(x)  where  p(x) = prod(x-[1:N])
%  and  g(x) = (p(-x) - (-x)^N)*(-1)^N .  The best candidate
%  has the minimum |beta| .  Use only  N = 18  in  MATLAB v. 7 .
Z = zeros(n-1,1) ;  B = [Z, Z] ;  N = [1:n] ;
G = abs(poly(N)) ;  G = G(2:n+1) ;  %... coefficients of  g(x)
G1 = G.*(n-N) ;  G1 = G1(1:n-1) ;   %... and of  g'(x)
for  k = 1:n-1
      z = fzero('f', [k+0.001, k+0.999], eps, [], n) ;
%...  For  MATLAB v. 7  omit these last two ~~~~~~~  arguments.
      Z(k) = z ;  p = prod(z-N) ;  p1 = p*sum(1.0./(z-N)) ;
      B(k,1) = -p/polyval(G,z) ;  B(k,2) = -p1/polyval(G1,z) ;
  end
```

= =

Results from `[B, Z] = betas(18)`:

```
              beta                       beta                             z
    7.38317832764355e-13      7.38317832764329e-13        17.7031785287344
   -1.18826070111930e-13     -1.18826070111929e-13        16.6358994232629
    3.78599411067942e-14      3.78599411067942e-14        15.5878476335195
   -1.92286101885325e-14     -1.92286101885327e-14        14.5482030390300
    1.42175942184190e-14      1.42175942184190e-14        13.5133543615164
   -1.46073512056147e-14     -1.46073512056148e-14        12.4815873694852
    2.03388743546567e-14      2.03388743546568e-14        11.4519209064207
   -3.79253582198457e-14     -3.79253582198455e-14        10.4237172002160
    9.44820694884286e-14      9.44820694884287e-14         9.39651627655296
   -3.16311667240260e-13     -3.16311667240261e-13         8.36995171255620
    1.44334556004274e-12      1.44334556004274e-12         7.34369877012772
   -9.19423774548570e-12     -9.19423774548568e-12         6.31743284329871
    8.48527106283812e-11      8.48527106283814e-11         5.29078229816549
   -1.20109201067620e-09     -1.20109201067620e-09         4.26325247166429
    2.85669177857461e-08      2.85669177857461e-08         3.23406142808911
   -1.34105658375384e-06     -1.34105658375384e-06         2.20166395679235
    0.00017686254475282       0.00017686254475282          1.16156230865673
»
```

These values ß seem accurate in all but their last digit or two because *all* pairs agree that closely despite that they reflect two utterly different ways to compute g'(x)/g(x) , one for `f(X, n)` and the other for `betas(n)` , each generating rounding errors in its own way different from the other.

Note how the ill-condition of a zero of `polyval(poly(1:18), x)` , as revealed by the loss of accuracy of its tabulated approximations computed by `roots(poly(1:18))` earlier, correlates with the smallness of a perturbation ß big enough to merge that zero with one of its neighbors.