Math 128a - Homework 2 - Due Feb 14 at the beginning of class

1) Complete Question 4 from Homework 1, which was postponed due to delayed availability of computer accounts.

Answer: Suppose $x > 0$. Here are two Matlab algorithms for computing e^{-x}:
Algorithm 1: Compute e^{-x} using a Taylor expansion

```
s = 1; t = 1; i = 1;
while (abs(t) > eps*abs(s))
      ... stop iterating when adding t to s does not change s
      t = -t*x/i;
      s = s + t;
      i = i + 1;
end
result1 = s;
```

Algorithm 2: Compute e^{-x} as $1/e^x$, using a Taylor expansion for e^x

```
s = 1; t = 1; i = 1;
while (abs(t) > eps*abs(s))
      ... stop iterating when adding t to s does not change s
      t = t*x/i;
      s = s + t;
      i = i + 1;
end
result2 = 1/s;
```

Part 1. Run these two algorithms for x = 1:20, tabulating the relative errors and number of iterations to converge for each.

Answer: The true answer was computed using the built-in exp() function in Matlab. The last column shows that the factor $3j$ in the error bound is an overestimate.

x	result1	# iters = j = i-1	rel error	3*j*eps* exp(2*x)	eps* exp(2*x)
1.0000e+00	3.6788e-01	1.9000e+01	3.0179e-16	9.3520e-14	1.6407e-15
2.0000e+00	1.3534e-01	2.4000e+01	4.1018e-16	8.7287e-13	1.2123e-14
3.0000e+00	4.9787e-02	2.9000e+01	1.6725e-15	7.7934e-12	8.9579e-14
4.0000e+00	1.8316e-02	3.4000e+01	1.7238e-14	6.7514e-11	6.6191e-13
5.0000e+00	6.7379e-03	3.8000e+01	2.1253e-13	5.5756e-10	4.8909e-12
6.0000e+00	2.4788e-03	4.2000e+01	7.0946e-13	4.5535e-09	3.6139e-11
7.0000e+00	9.1188e-04	4.6000e+01	1.7580e-11	3.6850e-08	2.6703e-10
8.0000e+00	3.3546e-04	5.0000e+01	6.5863e-12	2.9597e-07	1.9731e-09
9.0000e+00	1.2341e-04	5.4000e+01	1.4419e-10	2.3619e-06	1.4579e-08
1.0000e+01	4.5400e-05	5.8000e+01	3.0717e-09	1.8745e-05	1.0773e-07
1.1000e+01	1.6702e-05	6.2000e+01	2.8489e-08	1.4806e-04	7.9601e-07
1.2000e+01	6.1442e-06	6.6000e+01	1.5699e-07	1.1646e-03	5.8818e-06
1.3000e+01	2.2603e-06	6.9000e+01	1.6166e-06	8.9964e-03	4.3461e-05
1.4000e+01	8.3153e-07	7.3000e+01	4.2938e-07	7.0328e-02	3.2113e-04
1.5000e+01	3.0591e-07	7.7000e+01	2.3208e-05	5.4813e-01	2.3729e-03
1.6000e+01	1.1254e-07	8.1000e+01	5.8918e-05	4.2606e+00	1.7533e-02
1.7000e+01	4.1231e-08	8.4000e+01	4.0700e-03	3.2648e+01	1.2955e-01
1.8000e+01	1.5385e-08	8.8000e+01	1.0206e-02	2.5272e+02	9.5729e-01
1.9000e+01	8.8732e-09	9.1000e+01	5.8372e-01	1.9310e+03	7.0734e+00
2.0000e+01	5.6219e-09	9.5000e+01	1.7275e+00	1.4896e+04	5.2266e+01

x	result2	# iters = i-1 = j	rel error	(3*j+1)*eps	eps
1.0000e+00	3.6788e-01	1.8000e+01	1.5089e-16	1.2212e-14	2.2204e-16
2.0000e+00	1.3534e-01	2.3000e+01	2.0509e-16	1.5543e-14	2.2204e-16
3.0000e+00	4.9787e-02	2.7000e+01	2.7874e-16	1.8208e-14	2.2204e-16
4.0000e+00	1.8316e-02	3.0000e+01	3.7885e-16	2.0206e-14	2.2204e-16
5.0000e+00	6.7379e-03	3.3000e+01	3.8618e-16	2.2204e-14	2.2204e-16
6.0000e+00	2.4788e-03	3.6000e+01	0	2.4203e-14	2.2204e-16
7.0000e+00	9.1188e-04	3.9000e+01	8.3228e-16	2.6201e-14	2.2204e-16
8.0000e+00	3.3546e-04	4.1000e+01	1.6160e-16	2.7534e-14	2.2204e-16
9.0000e+00	1.2341e-04	4.3000e+01	2.1963e-16	2.8866e-14	2.2204e-16
1.0000e+01	4.5400e-05	4.6000e+01	2.9851e-16	3.0864e-14	2.2204e-16
1.1000e+01	1.6702e-05	4.8000e+01	0	3.2196e-14	2.2204e-16
1.2000e+01	6.1442e-06	5.0000e+01	4.1358e-16	3.3529e-14	2.2204e-16
1.3000e+01	2.2603e-06	5.3000e+01	0	3.5527e-14	2.2204e-16
1.4000e+01	8.3153e-07	5.5000e+01	3.8199e-16	3.6859e-14	2.2204e-16
1.5000e+01	3.0590e-07	5.7000e+01	3.4612e-16	3.8192e-14	2.2204e-16
1.6000e+01	1.1254e-07	5.9000e+01	0	3.9524e-14	2.2204e-16
1.7000e+01	4.1399e-08	6.1000e+01	7.9922e-16	4.0856e-14	2.2204e-16
1.8000e+01	1.5230e-08	6.3000e+01	2.1725e-16	4.2188e-14	2.2204e-16
1.9000e+01	5.6028e-09	6.5000e+01	1.4764e-16	4.3521e-14	2.2204e-16
2.0000e+01	2.0612e-09	6.7000e+01	2.0066e-16	4.4853e-14	2.2204e-16

Part 2. Prove that the relative error of result2 is, as you observe, bounded by $(3i-2)\epsilon$, i.e. very accurate. You may assume the error from terminating the Taylor expansion is smaller than round off error, and you may ignore terms proportional to ϵ^2. Confirm that $(3i-2)\epsilon$ bounds the relative errors in your table above.

Answer: Let $j = i-1$ be the number of passes through the loop. The analysis is somewhat like Horner's Rule, which we did in class. Let t_j be the exact value of t at the end of the j-th pass through the loop, and let $\widehat{t_j}$ be the computed value of t at the end of the j-th pass through the loop. We see that $t_0 = 1$, $t_1 = x$, $t_2 = x^2/2$ and in general $t_i = x^i/i!$ is the i-th term in the Taylor expansion of e^x. Because there are two roundoff errors at every step we get that $\widehat{t_i} = t_i \prod_{j=1}^{i-1}(1+\delta_i)(1+\delta_i') = t_i(1+r)$ where $|r| \le 2(i-1)\epsilon$. (If we do the multiplication $t*x$ first, then the $1+\delta_i$ is from $t*x$ and $1+\delta_i'$ is from $(t*x)/i$. We could do the division x/i or t/i first and get the same result. We may assume that i, being a small integer, is exact.) Next, consider summing the $\widehat{t_i}$ to get the final sum $\widehat{s_i} = fl(\sum_{j=0}^{i-1} \widehat{t_j})$. From the analysis in class, $\widehat{s_i} = \sum_{j=0}^{i-1} \widehat{t_j}(1+\delta_j'')$, where $|\delta_j''| \le (i-1)\epsilon$, so altogether $\widehat{s_i} = \sum_{j=0}^{i-1} t_j(1+r_j)$ where $|r_j| \le 3(i-1)\epsilon$. The final reciprocation $\widehat{s} = fl(1/\widehat{s_i})$ yields $\widehat{s} = 1/\sum_{j=0}^{i-1} t_j(1+r_j')$ where $|r_j'| \le (3i-2)\epsilon$. Ignoring the error from truncating the Taylor expansion we get

$$\begin{aligned} |e^x - \sum_{j=0}^{i-1} t_j(1+r_j')| &\approx |\sum_{j=0}^{i-1} t_j r_j'| \\ &\le \sum_{j=0}^{i-1} |t_j| \cdot |r_j'| \\ &\le (3i-2)\cdot\epsilon\cdot\sum_{j=0}^{i-1} t_j \\ &\approx (3i-2)\cdot\epsilon\cdot e^x \end{aligned}$$

or in other words $\sum_{j=0}^{i-1} t_j(1+r_j') = e^x(1+\eta)$ where $|\eta| \le (3i-2)\epsilon$, and so $\widehat{s} = e^{-x}(1+\eta')$ where $|\eta'| \le (3i-2)\epsilon$ as desired. Note that by counting rounding error more carefuly it may be possible to reduce the factor $3i-2$. But the important thing is that you get a multiple of ϵ that is not too large (unlike the next part).

Part 3. Prove that the relative error of result1 is bounded by $3(i-1)\epsilon e^{2x}$, i.e. it grows quickly with x, so that Algorithm 1 is much less accurate than Algorithm 2. You may make the same assumptions as before. Confirm that $3(i-1)\epsilon e^{2x}$ bounds the relative errors in your table above.

Answer: The identical analysis as above shows that the error is bounded by

$$\begin{aligned} |\widehat{s}_i - e^{-x}| &\approx |\sum_{j=0}^{i-1} t_j r'_j| \\ &\leq \sum_{j=0}^{i-1} |t_j| \cdot |r'_j| \\ &\leq (3i-3) \cdot \epsilon \cdot \sum_{j=0}^{i-1} |t_j| \\ &\approx (3i-3) \cdot \epsilon \cdot e^{|x|} \end{aligned}$$

Dividing through by e^{-x} yields the desired relative error bound. The difference between the two algorithms is that they both add a sum of quantities t_j with identical absolute values but different signs. In result2, the signs are all positive and the sum is accurate, but in result1, the signs alternate, the sum cancels (is much smaller than the largest t_j being added) and relative accuracy is lost.

Part 4. The computer implementation for e^x takes the same time for large and small arguments; i.e. it does not use a simple Taylor expansion, which would require more terms for larger arguments. Sketch an algorithm for e^x that does not take longer for large x. Use the fact that $e^x = 2^y$ where $y = x \cdot \log_2 e$, write $y = y_{int} + y_{frac}$ as a sum of an integer and a fraction less than 1, and use the fact that $2^y = 2^{y_{int}} \cdot 2^{y_{frac}}$ is to be rounded to a floating point number. How many term of a Taylor expansion of $2^{y_{frac}}$ are needed so that the remaining terms contribute less than ϵ to the relative error?

Answer: Here is a sketch of the algorithm to compute e^x: Ahead of time store values for $\log_2 e$ and $\ln 2$.

1. If $x = 0$ return 1, if $x < 0$ replace x by $-x$ and remember that you did this.
2. Let $y = x \log_2 e$, split y into $y_{int} + y_{frac}$ so that y_{int} is a positive integer and $0 \leq y_{frac} < 1$.
3. Compute $2^{y_{frac}}$ using the Taylor polynomial for $2^{y_{frac}}$ (as in Algorithm 2) but stop after 16 iterations (see below for justification of 16).
4. If we did not switch signs, return

$$(\text{computed value of } 2^{y_{frac}}) \times 2^{y_{int}}$$

   else return the reciprocal of that.

To see why 16 is the maximum number of steps we need, note that, after n terms in the Taylor polynomial for 2^z, where $z > 0$, the error is bounded in absolute value by:

$$\frac{z^{(n+1)}}{(n+1)!}(\ln 2)^{n+1} 2^z$$

since the $(n+1)$'th derivative of 2^z is $(\ln 2)^{n+1} 2^z$. We want to find n so that this error is bounded above by $\epsilon 2^z$, so that the relative error is bounded above by $\epsilon = 2^{-53}$. Dividing

both sides by 2^z, we want to find n such that:

$$\frac{z^{(n+1)}}{(n+1)!}(\ln 2)^{n+1} < 2^{-53}$$

and use $z = y_{frac} < 1$ and $\ln 2 < 1$ to get:

$$\frac{z^{(n+1)}}{(n+1)!}(\ln 2)^{n+1} < \frac{(\ln 2)^{n+1}}{(n+1)!}$$

and use Matlab to verify that if $n = 16$ then $\frac{(\ln 2)^{n+1}}{(n+1)!} < 2^{-53}$. I just ran the following line to find $m = n + 1$:

```
x=2^(-53);m=1;mfact=1;l2=log(2);
while(l2^n/mfact¿x),m=m+1;mfact=mfact*m;end;
m,mfact
```

2) In this question we will write a program to explore the sensitivity of roots of polynomials to perturbations in their coefficients. From the class homepage, download the matlab program polyperturb.m. Polyperturb takes an input polynomial specified by its roots r, and then adds random perturbations to the polynomial coefficients, computes the perturbed roots, and plots them. The inputs are

r = vector of roots of the polynomial
e = maximum relative perturbation to make to each coefficient of the polynomial
m = number of random polynomials to generate, whose roots are plotted

Try running this program on some of the inputs given below to help you understand what it does.

Based on the analysis done in class, where we computed the first term of the Taylor expansion of how much the roots can change when the coefficients are perturbed, modify this program to draw circles around each root, with radii equal to how much we expect the roots to change. The program should also return the values of the radii in the variable ebnd. Ideally, the circles should contain all the perturbed roots, as long as the perturbation size e is not too large. Note that in class, we only worried about the sensitivity to roundoff; now you will use the same analysis to analyze the effect of a deliberate perturbation of size e, which is generally much larger than roundoff. You should turn in your program. (Note: the analysis we did in class return an error bound of infinity for multiple roots; that is an acceptable answer for this homework. This bound can of course be made smaller by further analysis, but you do not have to do this, unless you want to.)

Next, run your program for the following inputs. In all cases choose m high enough that you get a fairly dense plot, but don't have to wait too long. m = a few hundred or perhaps 1000 is enough. You may want to change the axes of the plot if the graph is too small or too large. ("axis([xmin,xmax,ymin,ymax])" changes the limits of the axes to the 4 values shown. "axis('square')" makes the plotting window square, so that if xmax-xmin = ymax-ymin, circles appear as circles instead of ellipses, etc.)

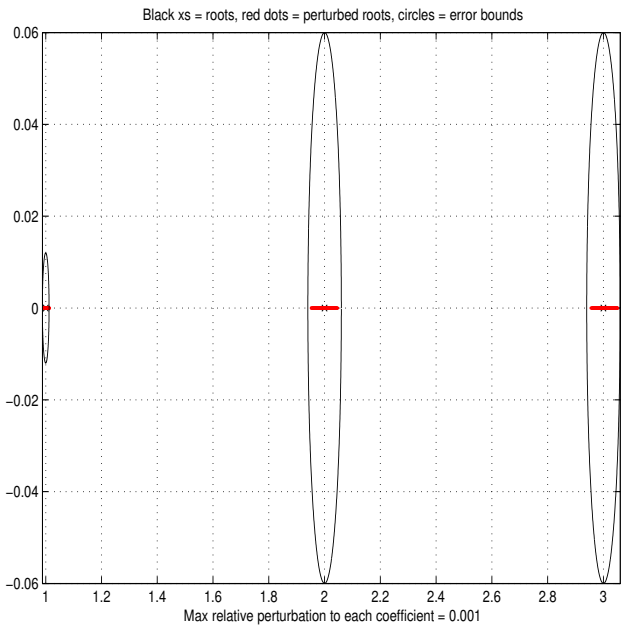
- r=[1,2,3]; e = 1e-6, 1e-4, (*)1e-3, 1e-2, 2e-2, 5e-2
  try "axis([2-100*e, 2+100*e, -100*e, 100*e])" when e is small, to see what happens around the root at 2 (or 1 or 3)
- r=[1,1.1,3]; e = 1e-6, 1e-4, 1e-3, 1e-2
- r=[1,2,2,3,3,3]; e = 1e-10, 1e-8, (*)1e-6, 1e-4, 1e-3
- r=(1:10); e = 1e-8, (*)1e-7, 1e-6
- r=(1:20); e = (*)1e-15, 1e-13
- r=[2,4,8,16, ... , 1024]; e=1e-4, (*)1e-3, 1e-2

You should turn in the values of ebnd (radii of the circles) and plots produced by your program corresponding to the values of e labeled by (*) above. Also briefly describe what you see in the plots, commenting on whether the circles do or do not contain the perturbed roots, (ie. whether your error bound is a good estimate of the actual error; it should be for small e).

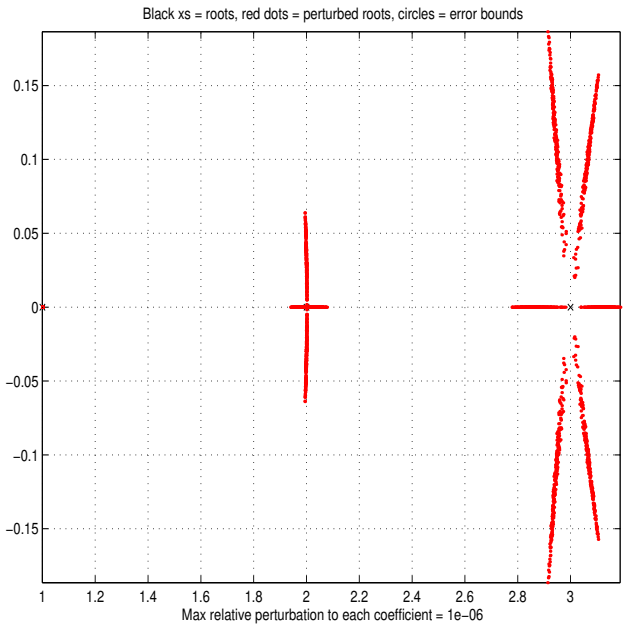
Answer: The program computing and plotting the error bounds is as follows:

```
% Matlab code polyperturb.m
% function ebnd = polyperturbANS(r,e,m)
%
% Form the coefficients of a polynomial specified by its roots,
%   and repeatedly add small perturbations to the coefficients,
%   plotting the resulting perturbed roots
%
% Inputs:
%   r = vector of polynomial roots
%   e = maximum relative perturbation to make to each coefficient
%   m = number of random polynomials to generate
%
% Output:
%   ebnd = error bounds for each root (to have been supplied by students!)
%   Plot of perturbed roots
%
function ebnd = polyperturbANS(r,e,m)
%
% Generate polynomial coefficients
p=poly(r);
%
% Generate m random polynomials and compute their roots
r1save=[];
for i=1:m,
%  Add random relative perturbation of at most e to each coefficient
   p1=p.*(ones(size(p))+e*2*(rand(size(p))-.5));
%  Compute and save perturbed roots
   r1=roots(p1);
   r1save=[r1save;r1];
end
%
%  compute error bound for each root
n = length(r);
% compute coefficients of derivative of polynomial
pprime = (n:-1:1).*p(1:n);
for i=1:length(r);
   ebnd(i) = e*polyval(abs(p),abs(r(i)))/abs(polyval(pprime,r(i)));
end
%
%  Compute min and max roots to determine axes for plotting
   for i=1:n,
      pltbnd(i) = ebnd(i);
      if isinf(ebnd(i)),
```

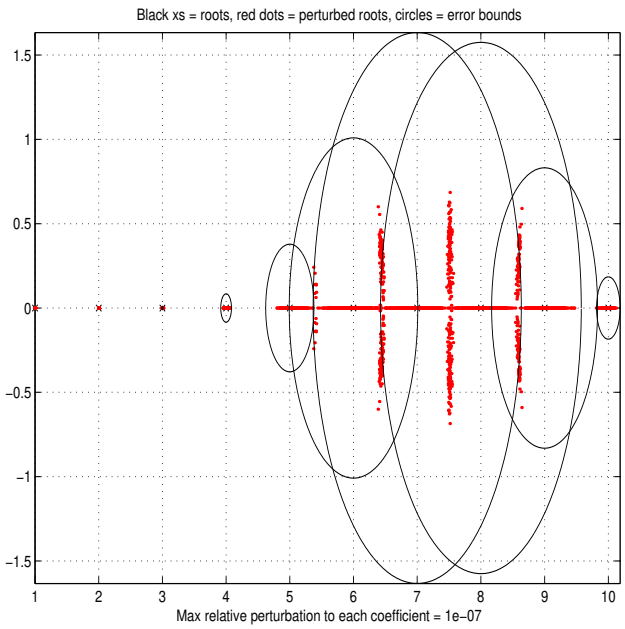
```
         pltbnd(i)=0;
      end,
   end
   minx = min(min([real(r1save);real(r')-pltbnd']));
   maxx = max(max([real(r1save);real(r')+pltbnd']));
   miny = min(min([imag(r1save);imag(r')-pltbnd']));
   maxy = max(max([imag(r1save);imag(r')+pltbnd']));
%
%  If all roots lie in the right halfplane, and
%  vary greatly in magnitude, use a semilog plot
figure(1), hold off, clf,
theta = 0:.01:2*pi;costheta = cos(theta);sintheta=sin(theta);
if (minx>0 & minx <= .002*maxx),
%  Plot unperturbed and perturbed roots
   semilogx(real(r),imag(r),'kx'), hold on
   semilogx(real(r1save),imag(r1save),'r.')
   for i=1:length(r);
     semilogx(real(r(i))+ebnd(i)*costheta,imag(r(i))+ebnd(i)*sintheta,'k')
   end
else
%  Plot unperturbed and perturbed roots
   plot(real(r),imag(r),'kx'), hold on
   plot(real(r1save),imag(r1save),'r.'), hold on,
   for i=1:length(r);
     plot(real(r(i))+ebnd(i)*costheta,imag(r(i))+ebnd(i)*sintheta,'k')
   end
end
%
% Adjust axes on plot to make it look nice
axis([minx,maxx,miny,maxy])
grid
title(['Black xs = roots, red dots = perturbed roots, circles = error bounds'])
xlabel(['Max relative perturbation to each coefficient = ',num2str(e)])
```



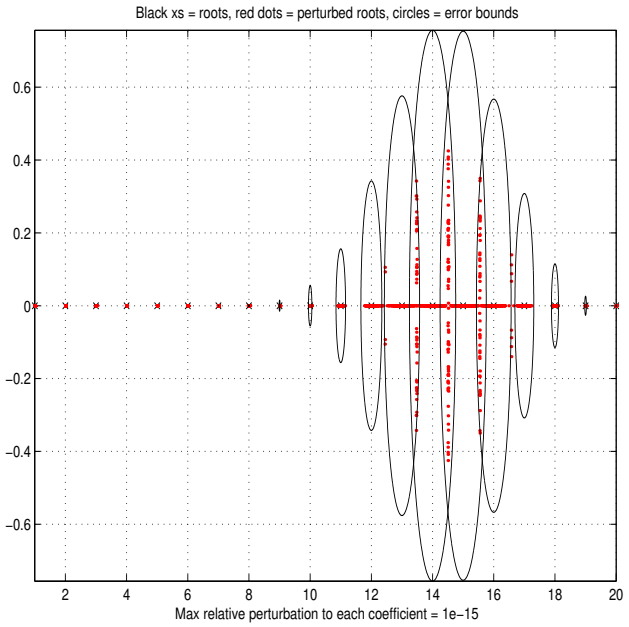
Exact roots are 1,2,3, e=1e-3, with error bounds 1.2e-2, 6e-2, 6e-2. Perturbed roots are within error circles, close to boundary (so that bounds are about as small as possible).



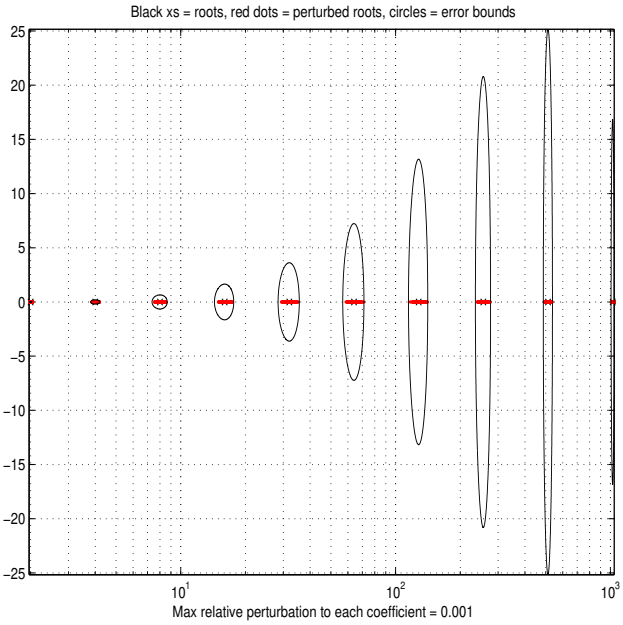
Exact roots are 1,2,2,3,3,3, e = 1e-6, with error bounds 1.44e-4, ∞, ∞, ∞, ∞, ∞. Error bounds are infinite for roots 2 and 3 since they are multiple root. Perturbed roots near 1 are within error circle, close to boundary.



Exact roots are 1,2,...,10, e=1e-7, with error bounds 1.10e-5, 5.94e-4, 1.03e-2, 8.41e-2, 3.78e-1, 1.01, 1.63, 1.58, 8.31e-1, 1.85e-1. Smallest and largest roots are less sensitive than middle roots, which can form multiple roots.



Exact roots are 1,2,...,20, e=1e-15, with error bounds 4.2e-13, 8.78e-11, 6.06e-9, 2.06e-7, 4.12e-6, 5.35e-5, 4.82e-4, 3.13e-3, 1.51e-2, 5.56e-2, 1.56e-1, 3.42e-1, 5.76e-1, 7.56e-1, 7.54e-1, 5.67e-1, 3.08e-1, 1.15e-1, 2.62e-2, 2.76e-3. Again, smallest and largest roots are less sensitive than middle roots, which can form multiple roots.



Exact roots are 2,4,8,16,...,1024, e=1e-3, with error bounds, 3.29e-2, 1.97e-1, 6.50e-1, 1.65, 3.62, 7.23, 1.32e+1, 2.08e+1, 2.52e+1, 1.68e+1

3) Determine whether or not the following two sequences converge quadratically or not:

Part 1: $a_n = 1/2^{2^n}$

Answer: Yes, it does converge quadratically to its limit $a = 0$, because $a_{n+1} = a_n^2$.

Part 2: $b_n = n^{-n}$

Answer: No, it does not converge quadratically to its limit $b = 0$. If it did, so that $b_{n+1} \le Cb_n^2$ for some $C > 0$, then we would have $b_n^2/b_{n+1} \ge 1/C > 0$, so that $\lim_{n\to\infty} b_n^2/b_{n+1}$ would be positive. However the limit is actually

$$\begin{aligned}
\lim_{n\to\infty} b_n^2/b_{n+1} &= \lim_{n\to\infty} n^{-2n}/(n+1)^{-n-1} \\
&= \lim_{n\to\infty} \left(\frac{n+1}{n}\right)^n \cdot \frac{n+1}{n} \cdot \frac{1}{n^{n-1}} \\
&= e \cdot 1 \cdot \lim_{n\to\infty} \frac{1}{n^{n-1}} \\
&= 0
\end{aligned}$$

4) Suppose we use the bisection algorithm to find the zero of a polynomial. Show that it is backward stable. In other words, prove that if bisection returns $[x_l, x_u]$ as the interval containing a root of the polynomial $p(x) = \sum_{i=0}^n c_i x^i$, then there is a another polynomial $\hat{p}(x) = \sum_{i=0}^n (c_i + \delta c_i) x^i$ where

- $|\delta c_i| \le 2n\epsilon |c_i|$, where $\epsilon = 2^{-53}$ in IEEE arithmetic (in other words, $\hat{p}$ is just slightly different from p), and
- $\hat{p}(x)$ does have a zero $\widehat{x}$ in the interval $[x_l, x_u]$.

Hint: Use the Intermediate Value Theorem.

Answer: At the end of bisection, the algorithm asserts that there are polynomials $p_u(x)$ and $p_l(x)$ such that $p_u(x_u) \cdot p_l(x_l) < 0$, and where $p_u(x) = \sum_{i=0}^n (c_i + \delta c_{u,i}) x^i$ with $|\delta c_{u,i}| \le 2n\epsilon |c_i|$, and $p_l(x) = \sum_{i=0}^n (c_i + \delta c_{l,i}) x^i$ with $|\delta c_{l,i}| \le 2n\epsilon |c_i|$. Now define the polynomial

$$\begin{aligned}
q(x,t) &= (1-t) \cdot p_l(x) + t \cdot p_u(x) \\
&= \sum_{i=0}^n (c_i + [(1-t) \cdot \delta c_{l,i} + t \cdot \delta c_{u.i}] x^i
\end{aligned}$$

and the function $X(t) = (1-t) \cdot x_l + t \cdot x_u$. Note that as t increases from 0 to 1, $q(x,t)$ changes from $q(x,0) = p_l(x)$ to $q(x,1) = p_u(x)$, and $X(t)$ increases from x_l to x_u. Finally, consider $r(t) = q(X(t), t)$. Then

$$r(0) \cdot r(1) = q(x_l, 0) \cdot q(x_u, 1) = p_l(x_l) \cdot p_u(x_u) < 0$$

In other words, $r(t)$ changes sign on the interval $[0, 1]$, so by the Intermediate Value Theorem $r(s) = q(X(s), s) = 0$ for some $0 < s < 1$. We claim $q(s, x) = \hat{p}(x)$ is the desired polynomial, with

zero $X(s) = \widehat{x}$ satisfying $x_l < \widehat{x} < x_u$. It is easy to see that the coefficients $c_i + [(1-s)\cdot\delta c_{l,i} + s\cdot\delta c_{u,i}] \equiv c_i + \delta c_i$ of $\hat{p}(x)$ satisfy

$$\begin{aligned} |\delta c_i| &= |(1-s)\cdot\delta c_{l,i} + s\cdot\delta c_{u,i}| \\ &\leq (1-s)\cdot|\delta c_{l,i}| + s\cdot|\delta c_{u,i}| \\ &\leq (1-s)\cdot 2n\epsilon|c_i| + s\cdot 2n\epsilon|c_i| \\ &= 2n\epsilon|c_i| \end{aligned}$$

as desired. $\hat{p}(\widehat{x}) = 0$ by construction, and it is easy to see that $x_l < X(s) = \widehat{x} < x_u$ as desired.

5) The mathematical function

$$f(x) = \begin{cases} \ln(1+x)/x & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases}$$

has all continuous derivatives for $x > -1$ Show that the obvious way to implement it in Matlab

```
function y = f1(x)
    if (x == 0)
        y = 1
    else
        y = log(1+x)/x;
    endif
```

is not backward stable. Hint: try evaluating it very near to 0. Does f1(x) = f(x+dx) where $|dx|/|x|$ is very small?

Answer: If $-2^{-54} < x < 2^{-53}$, so that $fl(1+x) = 1$, then $\log(1+x) = \log(1) = 0$ and $y = 0$, whereas the true value of $f(x)$ is very close to 1, and in fact always positive for $x > -1$.

Extra Credit (harder): show that the alternative below is backward stable. You may assume that the computed value of $\log(z)$ has very tiny relative error, near 2^{-52}.

```
function y = f2(x)
    z = 1+x;
    if (z == 1),
        y = 1;
    else
        y = log(z)/(z-1);
    end
```