Solutions to Assignment #7

Problems 6.1, exercise 13: Let $f(x) = \cosh(x)$. Let $x_0, x_1, \ldots, x_{22}$ be any 23 distinct nodes in $[-1, 1]$. Let $p(x)$ be the unique degree 22 polynomial with $p(x_j) = f(x_j)$ for all j. Then $p(x)$ has relative approximation error less than 5×10^{-16}, and in fact:

$$\frac{|p(x) - f(x)|}{|f(x)|} \le 3.813 \times 10^{-16}.$$

Proof. We apply Lagrange's remainder theorem (page 315). Note $f(x) = \cosh x$ has as many continuous derivatives as desired, and $f^{(23)}(x) = \sinh x$. Also, we know nothing detailed about x_j, but we do know that both x and x_j are in $[-1, 1]$, so $|x - x_j| \le 2$. If x is in $[-1, 1]$ then

$$|f(x) - p(x)| = \frac{|f^{(23)}(\xi)|}{23!}|x - x_0|\,|x - x_1|\,\ldots\,|x - x_{22}| \le \frac{|\sinh \xi|}{23!}2^{23} \le \frac{(\sinh 1)2^{23}}{23!}.$$

(Note that $\sinh x$ is increasing, so it takes its maximum value at the end of the interval.)

The relative error can now be approximated, because $f(x) = \cosh x \ge \cosh 0 = 1$, because $y = \cosh x$ has its minimum at $x = 0$:

$$\frac{|p(x) - f(x)|}{|f(x)|} \le \frac{(\sinh 1)2^{23}}{(\cosh 0)23!} = 3.813 \times 10^{-16}.$$

□

Problems 6.1, exercise 27: (*This is slightly easier than the last.*) The unique degree 12 polynomial $p(x)$ which interpolates $f(x) = e^{x-1}$ at 13 distinct points $x_0, x_1, \ldots, x_{12}$, all in the interval $[-1, 1]$, has the property that if x is any point in $[-1, 1]$ then

$$|f(x) - p(x)| \le \frac{2^{13}}{13!} = 1.316 \times 10^{-6}.$$

Proof. By Lagrange's remainder theorem, noting $f^{(13)}(x) = f(x) = e^{x-1}$, which has a maximum of $e^0 = 1$ on $[-1, 1]$,

$$|f(x)-p(x)| = \frac{|f^{(13)}(\xi)|}{13!}|x-x_0|\,|x-x_1|\,\ldots\,|x-x_{12}| \le \frac{e^{\xi-1}}{13!}2^{13} \le \frac{1 \cdot 2^{13}}{13!} = 1.316\times 10^{-6}$$

(*Since we do not know where the nodes are, we cannot do better than this estimate.*) □

Problems 6.1, exercise 37: The first thing I did was put the data in a readable file:

stampdata.m

```
% STAMPDATA  Just store the data.

t = [1885, 1917, 1919, 1932, 1958, 1963, 1968, 1971, 1974, 1978, ...
     1981+(3/12), 1981+(10/12), 1985, 1988, 1991, 1995, 1999, 2001]';
y = [   2     3     2     3     4     5     6     8    10    15 ...
             18           20    22    25    29    32    33    34]';
```

Next, here is a code that solves for the coefficients c_j of the Newton polynomial; it sets up a triangular matrix and uses built-in backslash to solve:

stamps.m

```
function cost = stamps(year)
% STAMPS  Fit high degree polynomial in Newton form through
%         stamp cost data.  Evaluates it at the input year.

stampdata  % fills t,y with data

ts = t - 1885;  % = [0 32 34 47 ... 106 110 114 116]

% build Newton matrix to find c_j in
% p(x) = c_1 + c_2 (x-0) + c_3 (x-0)(x-32) + c_4 (x-0)(x-32)(x-34)
%         + ... + c_18 (x-0)(x-32)...(x-114)
A = zeros(18,18);
A(1,1) = 1;
for i=2:18   % i = row index
  prodx = 1;
  for j=1:i  % j = col index
    A(i,j) = prodx;
    prodx = prodx * (ts(i)-ts(j));
  end
end
% to view checkable small chunk:  A(1:3,1:3)

c = A \ y;  % find coeffs: automatically recognizes triangular
            %   system and solves by forward substitution

% eval at input
x = year - 1885;
cost = c(1);
prodx = 1;
for j=2:18
  prodx = prodx * (x - ts(j-1));
  cost = cost + c(j) * prodx;
end
```

The degree 17 polynomial passing exactly through the data is a disaster. Here is one way to plot it, along with an actually useful thing, namely a degree 4 polynomial which closely fits the data:

```
>> stampdata
>> tf=1875:0.2:2010; zf=zeros(size(tf));         % fine grid for plotting
>> for j=1:length(tf), zf(j)=stamps(tf(j)); end
>> p4 = polyfit(t,y,4);
warning: dgelsd: rank deficient 18x5 matrix, rank = 4
>> plot(t,y,'o',tf,zf,tf,polyval(p4,tf))
>> axis([1875 2010 -5 50]), grid on
```

This produces Figure 1, with the degree 17 polynomial off-scale in many places.

Now we are asked for the prediction, from this degree 17 polynomial, of when we will hit price \$1. I interpret this as "find the first time after the last price increase in 2001

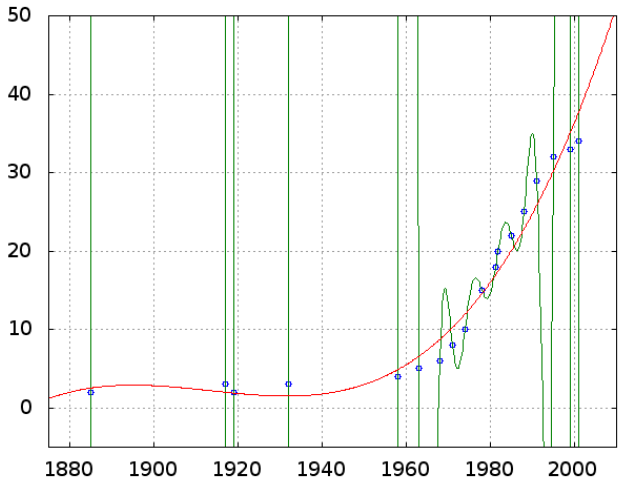

FIGURE 1. The lousy degree 17 polynomial goes through all the data but is not useful for interpolation, much less extrapolation. The least squares degree 4 (regression) polynomial might be useful.

when the price hits \$1." This is a job that can be done by bisection, which is already written, namely `bis.m`:

```
>> format long
>> stamps(2001)
ans =     33.9999999997599
>> f = @(x) stamps(x) - 100.0;
>> bis(2001,2001.1,f)
ans =     2001.02932726536
>> stamps(ans)
ans =     99.9999999999054
```

That is, the degree 17 polynomial "predicts" that the cost of a stamp is \$1 about eleven days (0.029 years) after it hit the price of 34 cents. Similar calculations give that the cost is \$10 a few months later. Dumb.

The degree 4 least squares (regression) polynomial would, not surprisingly, be more useful on this "real data" problem, even though it does not pass through the data.

Exercise 1: Recall the theorem given in class:

> **Theorem.** *Suppose f is in $C^2[a,b]$ and $a = x_0 < x_1 < \cdots < x_n = b$. If $L(x)$ is the piecewise-linear interpolant through the points $\{(x_j, f(x_j)\}$ then for each x in $[a,b]$,*
>
> $$|L(x) - f(x)| \le \frac{1}{2}\left(\max_{a\le x\le b}|f''(x)|\right)\left(\max_{j=0,\ldots,n-1}|x_{j+1}-x_j|^2\right).$$

This problem asks you to apply this theorem and Lagrange's theorem (page 315) and compare the results. In all cases the full interval is $[a,b] = [0,1]$ so for polynomial interpolation we have $|x - x_j| \le 1$.

(a) $n = 10$ degree polynomial interpolation:

$$|f(x)-p_{10}(x)| = \frac{|f^{(11)}(\xi)|}{11!}|x-0|\,|x-0.1|\,|x-0.2|\,\ldots\,|x-1.0| \le \frac{e^{-\xi}}{11!}1^{11} \le \frac{e^0}{11!} = \frac{1}{11!} = 2.5\times10^{-8}$$

piecewise-linear interpolation:

$$|f(x)-L(x)| \le \frac{1}{2}\left(\max_{0\le x\le 1}|f''(x)|\right)\left(\max_j |x_{j+1}-x_j|^2\right) = \frac{1}{2}\left(\max_{0\le x\le 1} e^{-x}\right)(0.1)^2 = \frac{1}{2}e^0(0.01) = 0.005$$

Clearly polynomial interpolation gets better fit with this many points on this function.

(b) $n = 10$ degree polynomial interpolation:

$$|f(x)-p_{10}(x)| = \frac{|f^{(11)}(\xi)|}{11!}|x-0|\,|x-0.1|\,\ldots\,|x-1.0| \le \frac{7^{11}|\cos(7\xi)|}{11!}1^{11} \le \frac{7^{11}}{11!} = 49.5$$

piecewise-linear interpolation:

$$|f(x)-L(x)| \le \frac{1}{2}\left(\max_{0\le x\le 1}|f''(x)|\right)\left(\max_j |x_{j+1}-x_j|^2\right) = \frac{1}{2}\left(7^2\right)(0.1)^2 = 0.245.$$

Here polynomial interpolation is struggling. Neither method gives great fit, but we know in advance that piecewise-linear is not completely unreasonable.

(c) $n = 25$ degree polynomial interpolation:

$$|f(x)-p_{25}(x)| = \frac{|f^{(26)}(\xi)|}{26!}|x-0|\,|x-0.04|\,\ldots\,|x-1.0| \le \frac{7^{26}|\sin(7\xi)|}{26!}1^{26} \le \frac{7^{26}}{26!} = 2.32\times10^{-5}$$

piecewise-linear interpolation:

$$|f(x)-L(x)| \le \frac{1}{2}\left(\max_{0\le x\le 1}|f''(x)|\right)\left(\max_j |x_{j+1}-x_j|^2\right) = \frac{1}{2}\left(7^2\right)(0.04)^2 = 0.0392.$$

Here polynomial interpolation has better accuracy than piecewise-linear.

(d) On the one hand, piece-wise linear interpolation accuracy is boring, and knowing the subinterval length determines the error. On the other hand, for $f(x) = \sin(7x)$ the error in n degree polynomial interpolation is more interesting. The difference between 11 points and 26 is very substantial. The error for polynomial interpolation comes from an interplay of competing forces, and the 7^{n+1} term for n degree polynomial interpolation competes with the magnitude of $(n+1)!$ to determine the error size.

Exercise 2: **(a)** The degree 4 version works fine. I use `polyfit` only for brevity:

```
>> p = @(x) (x-1).*(x-2).*(x-3).*(x-4);
>> x = [0.1 0.9 1.4 3.14159, 4.1]; y = p(x);
>> polyfit(x,y,4)
ans =
    1.00000  -10.00000   35.00000  -50.00000   24.00000
```

Note that the *exact* standard form of the polynomial is

$$p(x) = x^4 - 10x^3 + 35x^2 - 50x + 24,$$

so, by continuing the calculation I estimate the error as

```
>> max(abs(ans - [1 -10 35 -50 24]))
ans =  4.97379915032070e-14
```

(b) To do this we need to evaluate the polynomial in its compact form:

wilkp.m

```
function z = wilkp(x)
% WILKP  Do a good job of computing values of the Wilkinson
%        polynomial, by evaluating it in factored form.

z = ones(size(x));
for i=1:20
  z = z .* (x - i);
end
```

The method of my choice, to find the coefficients of the same polynomial but in standard form, is again `polyfit`, which internally is doing the Vandermonde method.

```
>> x = 0:20 + 0.141;  y = wilkp(x);
>> AA = polyfit(x,y,20);
warning: matrix singular to machine precision, rcond = 1.7636e-32 ...
```

(c) As noted, I have written `wilkinson.m` to report the known coefficients in standard form, so we continue the calculation, comparing computed to exact:

```
>> wilkinson  % get stored coefficients a(j)
>> max(abs(AA - a))
ans =  1.38037597536407e+19
```

This is a huge error.

(Extra Credit) Running the following program shows that disaster happens suddenly. Degree 11 interpolation is essentially o.k. but degree 12 fails utterly. (*I* don't know why the transition is so sudden.)

wilkextra.m

```
% WILKEXTRA  Solution to extra credit for Exercise 2, A#7.

errorzero = [];
for m=4:20
  j = 0:m;
  x = j + 0.141;
  y = x - 1;
  for k = 2:m
    y = y .* (x - k);
  end
  a_computed = polyfit(x,y,m);       % generates lots of warnings
  exact = (-1).^m * factorial(m);    % exact constant term in poly
  errorzero = [errorzero abs(a_computed(end)-exact)];
end
semilogy(4:20, errorzero,'o','markersize',14), grid on
xlabel('degree'), ylabel('error in constant term')
```