Math 126 Numerical PDEs, Winter 2012: Homework 2

due Monday 9am Jan 23

You'll need to leave some time for the coding for 4, 5, 6, which grow the same code. Tips: For plots, decide whether a log or linear axis is most useful. Matlab filenames should match function names.

1. Explain for each part why the two code versions give different answers, and why one is more accurate (that is, closer to the true answer) than the other. [Hint: first run them; (a) gives you a clue to (b)!]

   (a) way I: `a = (1 + 3.4e-16) - 1.1e-16; a-1`
   way II: `a = 1 + (3.4e-16 - 1.1e-16); a-1`

   (b) way I: `x = 0.999; a = 0; for j=1:60000, a = a-x^j/j; end; a-log(1-x)`
   way II: `x = 0.999; a = 0; for j=60000:-1:1, a = a-x^j/j; end; a-log(1-x)`
   You can check that this Taylor series has converged as accurately as it can to $\ln(1-x)$ for $x = 0.999$, *i.e.* that including more terms doesn't fix the problem. (By the way, summing is a terrible way to evaluate a slowly-converging series; there are much better acceleration methods...)

   What is your conclusion about the best way to sum a list of numbers in floating-point arithmetic?

2. For the following problems, the algorithm stated is implemented on a machine obeying the floating-point axioms. Deduce whether the algorithm is *backward stable*, *stable*, or *unstable*. [NLA 15.1]

   (a) $f(x) = 2x$ computed via $x \oplus x$.
   (b) $f(x) = 1 + x$ computed via $1 \oplus x$.

3. Show that computing eigenvalues of a symmetric matrix is numerically *unstable* if it is done by evaluating the characteristic polynomial $\det(A - \lambda I)$ then solving for its roots. We may restrict to 2-by-2 diagonal matrices, for which the problem data is the diagonal entries, and the answer the eigenvalues (a trivial problem!) i) Show analytically that there is an $O(\varepsilon_{mach})$ perturbation of the polynomial coefficients from those in the case $A = I$, that leads to a much larger (how large?) change in the roots. ii) Thus explain why any algorithm which passes through the above step is not stable according to our definition. [Excellent stable algorithms do exist to compute eigenvalues; see [NLA]].

4. Make a Matlab function to evaluate Lagrange basis functions $l_k(x)$ which has the following interface, *i.e.* begins as follows (or similar if you use a different language).

```
function l = lagrange(x, xj)
% LAGRANGE - evaluate all lagrange poly's at x, ie l_k(x) for k=0...n
%
% Inputs: x is a single ordinate, and xj is a row vector of nodes x_0,...,x_n
% Outputs: l is a column vector containing the values l_0(x),...,l_n(x)
```

   By using `.\`, `prod`, etc, you should only need to write explicitly one loop, the one over the basis function index (but note the effort is still $O(n^2)$). Now write a separate script which sets up $n = 12$ equally-spaced nodes x_j with $x_0 = -1$ and $x_n = 1$, and calls your function to plot all Lagrange basis functions over $[-1, 1]$ on the same axes [Hint: if `y` is a rectangular array then `plot(x, y, '-');` plots each row of `y` against the x-values in `x`]. Add the nodes as `'*'` symbols along your x-axis. BONUS: How does $\sup_{x\in[-1,1]} |l_{n/2}(x)|$ appear to be blowing up vs n ?

5. Graph the interpolation error $E_n(x) := f(x) - (L_n f)(x)$ vs x for $n = 25$ for the functions,

   (a) $f(x) = \sin(2e^x)$, which is an entire (analytic) function.

   (b) $f(x) = (1 + 25x^2)^{-1}$, which illustrates the Runge phenomenon.

   Now for the first function make a plot of the sup norm of the interpolation error in $[-1, 1]$ vs n, up to $n = 40$. You should have exponential convergence up to some point. BONUS: explain the cause of the new behavior for $n > 25$.

6. Tweak your codes in the previous two questions to instead use Chebychev nodes $x_j = -\cos(\pi j/n)$, for $j = 0, \dots n$, and describe the changes. In particular: what is $\sup_{x\in[-1,1]} |l_k(x)|$ (roughly)? What is the maximum error now in #6b? (Does the Runge phenomenon persist? Test a sequence of n to find out.) Is the best acheivable interpolation error improved?