Due: September 16, 2016

MATH 320: HOMEWORK 2

Please read through sections 4.1 - 4.3 in the textbook. Then answer the following questions. Please submit all code and output with brief descriptions of what you are doing.

(1) Prove that a finite base-n representation is unique. In particular, show that if a number has a finite base-n representation, any other finite base-n representation must be the same one.

Structure of the Proof: First we take two base-n representations. We show that if they don't have the same largest power of n, they are not equal. Then, we show if they don't have the same largest coefficient, they are not equal. Then, we show that if they differ in *any* term, they are not equal. This means that if they represent the same number, they must be the same representation.

Suppose there are two base-n representations of X:

$$X = \sum_{i=-m}^{r} a_i n^i = \sum_{i=-p}^{q} b_i n^i.$$

$$0 \leq a_i \leq n-1, \qquad 0 \leq b_i \leq n-1.$$

Consider the largest terms, $a_r n^r$ and $b_q n^q$, such that $a_r \neq 0$ and $b_q \neq 0$. Suppose $r > q$. Then:

$$\sum_{i=-p}^{q} b_i n^i < \sum_{i=-\infty}^{q} (n-1)n^i$$

$$= (n-1)n^q \sum_{i=0}^{\infty} (n^{-1})^i = (n-1)n^q \frac{n}{n-1} = n^{q+1}.$$

Since $r > q \implies r \geq q+1 > \sum_{i=-p}^{q} b_i n^i$, the two representations cannot be equal.

If $q > r$, then by the same logic, the numbers cannot be equal.

If $q = r$, then compare a_r and b_q. If $a_r > b_q$ then subtract $b_q n^q$ from both representations; they should still be equal, but we have reduced to the case $q > r$.

If $a_r = b_q$, then subtract $a_r n^r = b_q n^q$ from both representations. Repeat until the leading terms are distinct, which will happen if the representations are distinct. Then, we reduce again to the other cases.

(2) (Problem 4.4) For computers, the machine epsilon ϵ can also be thought of as the smallest number that when added to one gives a number greater than 1. An algorithm based on this idea can be developed as

Step 1: Set $\epsilon = 1$.

Step 2: If $1 + \epsilon$ is less than or equal to 1, then go to Step 5. Otherwise go to Step 3.

Step 3: $\epsilon = \epsilon/2$.
Step 4: Return to Step 2.
Step 5: $\epsilon = 2 \times \epsilon$.

Write your own M-file based on this algorithm to determine the machine epsilon. Validate the result by comparing it with the value computed with the built-in function `eps`.

The following code implements a `while` loop that returns the machine epsilon.

```
e = 1;
while (1 + e > 1)
    e = e/2;
end
e = 2*e;
```

This value is the smallest binary number that when added to 1 is not rounded off. The result should be: $2.220446049250313 \times 10^{-16}$ which is the decimal form of 2^{-52}.

(3) (Problem 4.11) The Maclaurin series expansion for $\cos x$ is:

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

Starting with the simplest version, $\cos x = 1$, add terms one at a time to estimate $\cos(\pi/4)$. After each new term is added, compute the true and approximate percent relative errors. Determine the true value. Add terms until the absolute value of the approximate error estimate falls below an error criterion conforming to two significant figures.

For this solution, we use the formulation for the n-th term of the series given by $(-1)^n x^{2n}/(2n)!$. We design a `while` loop which at each step, computes the next term of the Taylor series, updates the approximation, and then computes error. The error at each step is added to an array with two rows: first row gives the true relative error (`tre`), and second row gives the approximate relative error (`are`).
Because we want our answer to conform to two significant figures, we give the loop a stopping criterion of `are` $\leq .05$.

```
x = pi/4;
y = cos(pi/4);
cosApprox = 0;
n = 0;
error = zeros(2,0);
are = 1; tre = 1;
while (are > .05)
    term = (-1)^(n)/factorial(2*n)*x^(2*n);
    cosApprox = cosApprox + term;
```

```
    tre = (cosApprox - y)/y;
    if n > 0
        are = abs(term)/cosApprox;
    end
    error = [error [tre; are]];
    n = n + 1;
end
```

This outputs an error array with (rounded) values output below. The last approximate error value is below the desired .05.

```
error =

    0.4142    0.0220    0.0005
    1.0000    0.4460    0.0224
```

The value of the second-order approximation is 0.71.