CSC 270 midterm solutions

24 October 2001

1. [5 marks] The IEEE floating-point format has a sign bit, mantissa bits, and exponent bits.

The meaning of such a number, if the exponent bits do not represent one of the special signal values, is $\pm$ 1.{mantissa} $\times$ 2**({exponent}−127), where ‘‘**’’ represents exponentiation.

This question concerns identifying the value represented by a particular value in IEEE floating-point. Our number has a sign bit of zero (meaning that the value is greater-than-or-equal-to zero), a mantissa field of 0100000000000 (thus the mantissa value is 1.01_2, as the 1 before the binary point is implicit), and an exponent field of 01111100 (these eight bits represent the number 124).

What is the value represented? You do not have to simplify your answer, but you must represent it in usual, base ten, everyday terms. (That is, you can leave in something like 2^{58}, but that ‘‘2’’ and ‘‘58’’ must be expressed in base ten.)

Answer: $\frac{5}{32}$, or $1\frac{1}{4}\times 2^{-3}$

2. [5 marks]

2a. State any one source of error in assignment one’s computation of the sine function by the summation of $(-1)^i \frac{x^{2i+1}}{(2i+1)!}$.

Obvious answers: truncation error, or round-off error.

2b. What do we do to limit the error you identified in 2a? Explain in a few words.

Answer for truncation error: Since the truncation error is an increasing function of x, the range-reduction algorithm gives us a smaller x and hence smaller truncation error for a given number of terms. Another answer is that we figure out an error bound as a function of the number of terms, and then we use enough terms to make the truncation error acceptably low.

The answer for round-off error is harder. We mostly didn’t deal with round-off error in assignment one. However, in some cases the range-reduction algorithm also helped, in that for large x, due to the range-reduction algorithm we could get by with a reasonable number of terms of the series and thus we performed fewer operations (even including the range-reduction algorithms), thus typically getting less cumulative round-off error.

3. [5 marks] In comparing algorithms for computing the value of π, would you base your decision on the absolute error or the relative error, and why?

Answer: It doesn’t matter, because the desired actual value is always π. So using relative error just divides all the error amounts by π and the ranking doesn’t change.

4. [10 marks] There is a math library function *sqrt* which calculates square roots. It takes a single parameter of type `double` and returns type `double`. Write a complete C program which takes no input, and outputs just one number which is the value of:

$$\sum_{i=2}^{10} \sqrt{\frac{123}{i}}$$

You do not need to include comments, but note that ‘‘a complete C program’’ contains #includes, etc. That is, your program as written on this test paper must compile and run with no additions.

(over)

Answer to question 4:

```
#include <stdio.h>
#include <math.h>

int main()
{
    double sum = 0;
    int i;

    for (i = 2; i <= 10; i++)
        sum += sqrt(123.0 / i);
    printf("%g\n", sum);

    return 0;
}
```

It is essential to ensure that the division happens in floating-point. Integer division yields the wrong answer.

5. [10 marks] Recall Newton's method, with which we found a square root of y by finding zeroes of $f(x) = x^2 - y$. Each successive ''guess'' is calculated as $x - \frac{f(x)}{f'(x)}$, where x is the previous guess. Write a C function which uses Newton's method to find a zero of the function $f(x) = x^4 + x + 1$, to an error tolerance of 10^{-10}.

Your function will have no parameters and will return type `double`.

You needn't simplify the main formula in your function.

(The derivative of $x^4 + x + 1$ (with respect to x) is $4x^3 + 1$.)

Answer:
(the file must #include <math.h>, but this function might be thrown in with others and math.h might already be included)

```
double newton()
{
    double x = -0.7;
    double xsq /* x squared */,
           fx /* f(x) */;

    while (1) {
        xsq = x * x;
        fx = xsq * xsq + x + 1;
        if (fabs(fx) < 1e-10)
            break;
        x -= fx / (4 * x * xsq + 1);
    }

    return x;
}
```