CS2035 - Assignment 1 - 2018

IEEE 754 Floating Point Numbers

Out: January 14th, 2018

In: January 28th, 2018 at 11:55 pm via Owl

Introduction

This MATLAB assignment requires you to write four (4) MATLAB functions, each contained in its own script file. The first two convert a signed integer to 8 bit decimal (`int2bin8.m`) and convert an 8 bit decimal to a signed integer (`bin2int8.m`). The second two functions convert a decimal number to a 32 bit binary floating-point representation (`dec2bin32.m`) and a 32 bit binary string encoding a floating-point number to a decimal number (`bin2dec32.m`).

You will learn about:

- binary representation of an 8-bit signed integer;
- IEEE 754 binary representation of a single precision (32 bit) floating-point number;
- **if-then-else** control statements;
- substrings;
- creating MATLAB functions.

This assignment is worth 8% of the course mark.

IEEE 754 Floating Point Numbers

Introduction

In 1985 the Institute of Electrical and Electronics Engineers (IEEE) established a technical standard for floating-point computation, which ensures that (to a reasonable extent) the implementation of of floating-point arithmetic on different systems is reliable and code is portable. The full standard is very complex, so we will just learn about the basics of representing floating-point numbers in this standard.

We are all familiar with scientific notation for decimal numbers, which represents a number as some decimal number between 1 and $9.\overline{9}$ (called the *mantissa*) multiplied by 10 (the base) to an integer exponent. The mantissa is always restricted to a certain number of decimal digits (determining the precision of the representation of a number). Thus, 10π can be represented approximately in scientific notation as

$$3.141592 \times 10^1, \tag{1}$$

where the mantissa here has a precision of 6 decimal digits.

The IEEE 754 standard for binary representation of floating point numbers works in much the same way, but using 2 rather than 10 for the base, and binary for the mantissa and exponent. To move in this direction, if we use base 2 and a binary mantissa (using decimal for the exponent for now), 10π can be represented approximately in scientific notation as

$$1.111101_2 \times 2^4, \tag{2}$$

where the mantissa here has a precision of 6 binary digits. The decimal equivalent of this number is $31.25 = 2^4 + 2^3 + 2^2 + 2^1 + 2^0 + 2^{-2}$, which is less than but close to the decimal value 31.41592 in (1) above. To recover the precision of (1), we actually require a precision of 17 binary digits! But what we have here is not yet an IEEE 754 floating-point number.

Single Precision Floating-Point Numbers

For this assignment we will focus on the representation of *single precision* floating-point numbers. Such numbers use 32 binary digits to represent a floating-point number. This breaks down into 1 bit for the sign (0 for positive, 1 for negative), 8 bits for the signed exponent (giving values between -126 and $+127$), and 24 bits for the mantissa, i.e., 23 digits after the decimal place (precision of 23 binary digits).

You will notice that $1 + 8 + 24 = 33$, so you might think I have made an error. In fact, this is correct. The reason is that in binary the integer part of the mantissa is always 1, so that it does not need to be stored in memory. In decimal, the integer part of the mantissa can be any number from 1 to 9, but in binary the only possibility is 1. Thus, we only need 23 bits to represent a 24 bit mantissa in binary.

We will illustrate the 32 bit single float representation by writing 10π in binary. To do this we need the sign bit, which is zero here, the binary representation of the exponent, and the full binary mantissa. So we start with the sign bit, giving us

sign	exponent	mantissa
0	· · · · · · · ·	· ·

From (2) we already have part of the mantissa, so let's fill that in:

sign	exponent	mantissa
0	· · · · · · · ·	1 1 1 1 0 1 · · · · · · · · · · · · · · · ·

Notice that I only copied the six digits after the decimal point because *we don't need to store the leading* 1.

8 Bit Signed Integers

Now for the exponent. We see from (2) that the exponent is 4. To represent this as an 8 bit *signed* integer we need to apply a *bias*. With 8 bits we can represent the numbers 0 to 255. To get negative numbers we subtract the bias, which here will be $127 = 2^7 - 1$. Subtracting the bias gives us numbers from -127 to 128. Now, in IEEE 754, the boundary values -127 and 128 are reserved for special quantities. **For normal single precision floating-point numbers the exponent takes values from** -126 **to** 127.

To find the binary representation of a signed decimal integer, we need to add the bias (127) and compute the binary value of the result. So for -126, adding the bias gives 1, which in 8 bit binary is just 0000 0001. Similarly, for $+127$ adding the bias gives 254, which is 1111 1110 in 8 bit binary. For 0, adding the bias yields 127, which is 0111 1111 in 8 bit binary. For $+4$, the exponent in (2), we add the bias to get $131 = 128 + 2 + 1$, which is 1000 0011. Thus, we can now fill in the place of the binary exponent:

sign	exponent	mantissa
0	1 0 0 0 0 0 1 1	1 1 1 1 0 1 · · · · · · · · · · · · · · · ·

Computing A Binary Mantissa

To fill in the mantissa, we can use the same algorithm as presented in class to divide the decimal number by successively smaller powers of 2, counting a 0 if the power of 2 does not divide the

number, and counting a 1 if the power of 2 does divide it, and continuing the process with the remainder, stopping when all of the bits have been filled. When we do this for 10π, we can fill in the rest of the mantissa to obtain

sign	exponent	mantissa
0	1 0 0 0 0 0 1 1	1 1 1 1 0 1 1 0 1 0 1 0 0 1 1 1 0 1 1 0 0 0 1

We therefore now have the IEEE 754 single precision representation of 10π. Expressed as a binary string, this is 0100 0001 1111 1011 0101 0011 1101 0001, giving 4 bytes of data.

For most numbers, the exponent takes some value between -126 and 127 inclusive, as was pointed out above. The only exceptions to this we consider are 0 and numbers that are larger than can be represented in the system. We will treat any number with absolute value smaller than 2^{-126} as floating-point zero `0`, which has the binary string

$$\texttt{0} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$$

2^{128} and larger numbers cannot be approximated in the single precision number system and are represented by `Inf`, where

$$\texttt{Inf} = 0111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000.$$

Numbers -2^{128} or less similarly are represented by `-Inf`, where

$$\texttt{-Inf} = 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000.$$

Part I: Converting an Integer to Binary and Back

For the first part of this assignment you will write a MATLAB function `int2bin8` to convert an integer x in the range $-127 \leq x \leq 128$ into its binary representation, and a second function `bin2int8` to convert from binary to integer.

To get you started, the following pseudocode gives you the basic structure of `int2bin8`:

```
function s = int2bin8(x)
% int2bin8:
% Explanation of what the function does
is x in the range [-127,128]?
s = '';
num = x + bias
for i from 7 to 0 decreasing by 1
    compute remainder of num and 2^i
    concatenate '0' or '1' to s accordingly
    reset the value of num
end
```

For the `bin2int8` function, which takes an 8 bit binary string `s` as input and outputs an integer `x`, your function should start with

```
function x = bin2int8(s)
% bin2int8:
% Explanation of what the function does
```

You will then need to ensure that the input is indeed an 8 bit binary string (for which the `strlength` and `assert` functions will be useful), check the bits of the input for which powers of 2 to add to the result, and apply the bias to compute the decimal value for the output `x`.

Part II: Converting a Decimal to a Single Precision Float and Back

For the second part of the assignment, you will use your code from part I to write a MATLAB function `dec2bin32` to convert a decimal number to a 32 bit float, and a second function `bin2dec32` to convert a 32 bit float to its decimal value.

For this part, there are three things that need to be kept track of: the sign; the exponent; and the mantissa. For `dec2bin32` the algorithm is the following

```
function s = dec2bin32(x)
% dec2bin32:
% Explanation of what the function does
is abs(x) < 2^126? (zero case)
is abs(x) >= 2^128? (+/- Inf case)
out = '';
% set sign bit
check the sign of x and set the appropriate sign bit of s
% set exponent bits
find the first power of 2 smaller than x to determine the exponent e
convert e to binary8 using your int2bin8 and concatenate the result to s
% set mantissa bits
num = remainder of x and 2^e  % extract 'hidden bit' portion of mantissa
for i from 1 to 23 by 1
    rem = remainder of num and 2^(e-i)
    concatenate '0' or '1' to s accordingly
    reset the value of num
end
```

Note that if you hit one of the special cases at the beginning, the execution of the function should end and return to the calling function. For finding the value of the exponent `e`, you may find the `log2` and `floor` functions useful.

For the `bin2dec32` function, which takes a 32 bit binary string as input and outputs a decimal number `x`, your function should start with

```
function x = bin2dec32(s)
% bin2dec32:
% Explanation of what the function does
```

You need to ensure that the input is indeed a 32 bit binary string, extract the sign bit, the exponent bits and the mantissa bits, convert the exponent to decimal, write a loop to add the powers of 2 as specified by the exponent and mantissa and then apply the correct sign. Remember to take into account the hidden bit of the mantissa!

Part III: Program Validation

Write a fifth MATLAB script file `testConversions.m` that calls `dec2bin32(x)` for a list of decimal numbers, each individually denoted as `x`, calls `bin2dec32(s)` for a list of binary strings corresponding to single precision floating-point numbers, each individually denoted as `s`, and calls `bin2dec32(dec2bin32(x))` and `dec2bin32(bin2dec32(s))` for these lists of decimal and binary floating-point numbers. Do you get the expected results?

Submitting the Assignment

To complete the assignment, make a new folder in your OWL Drop Box called **A1** and place in that folder your five (5) MATLAB script files together with the file `testConversions.pdf` obtained by running MATLAB's `Publish` on `testConversions.m`. **You must include the pdf output to receive full marks for your assignment.**