Jonathan W. Valvano

(10) Question 1. Let **x(t)** be the input signal interfaced to a 12-bit ADC.

Part a) The amplitude is the

sqrt(Re[y(5)]^2+Im[y(5)]^2)= 0.5V.

Since some FFT functions scale differently from the FFT function we used in lab, I will accept as correct answers either 0.5V/128, or 0.5V/2 as well. The phase is

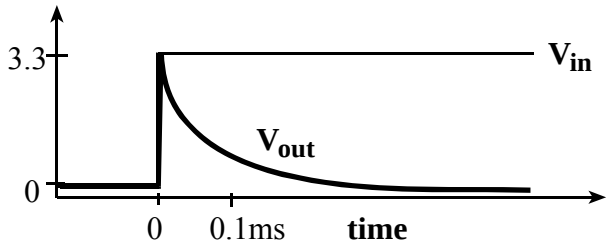
arctan(Im[y(5)]/Re[y(5)])= π/2.

The frequency is 5*10kHz/256 = 195.3 Hz. The x(t) = 0.5V cos(2π (195.3)t+ π/2)

Part b) k= 256-5 = 251 will also be nonzero, the complex conjugate of y(5). y(251) = -0.5j, or the real part of **y(251)** is zero, and the imaginary part of **y(251)** is -0.5 V.

(20) Question 2. This is a one-pole passive HPF with a time constant, τ, of RC = 100μs.

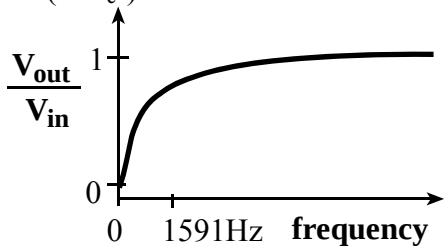
Part a) $\mathbf{V_{out}} = 3.3e^{-t/\tau}$.



Part b) The capacitor impedance is 1/(j2πf C). We use the resistor-divider problem (Ohm's Law) to find the gain as a function of frequency.

V_{out}/V_{in}=R/(R+1/(j2πf C))=(j2πf RC)/((j2πf RC)+1)

Let f_c be the filter cutoff = 1/(2π RC)=1591 Hz. The gain is the magnitude |(j f/f_c)/(j f/f_c +1)| which equals (f/f_c)/sqrt((f/f_c)^2 +1) or f/√(f^2+f_c^2)



I believe solutions to simple RC circuits like this should be in the short-term memory of all biomedical, electrical, and computer engineers.

(25) Question 3. Implement **y(n) = 0.3333333333x(n) – 0.2222222222y(n-1)**

Part a) Memory is a scarce resource. Engineers must learn to conserve it. We can not save every ADC sample. The idea of a digital filter is to pass it a stream of data, for each input to the filter there is one output. In order to implement the filter all we need is the previous filter output. So, we need a private 16-bit variable to store the old filter output

```
short static y;
```

Part b) The range of **`short`** numbers is -32768 to +32767, so when multiplying the input, the constant must be less than 32676/4095 = 8. Approximate the first constant by 3/9 and the second constant by 2/9. Divide last to reduce the effect of dropout.

```
short IIR(short input){  // input is the new ADC
  return y = (3*input-2*y)/9;
}
```

The simple way to prove this filter can not overflow is to assume the magnitude of all input and output numbers will be less than 4096. 3*4096-2*(-4096) < 32768. We know the input numbers are less than 4096, but to know the range of output numbers, we will need to know the gain as a function of frequency and find the maximum gain. The DC gain of this filter can be calculated by solving filter for constant input/output

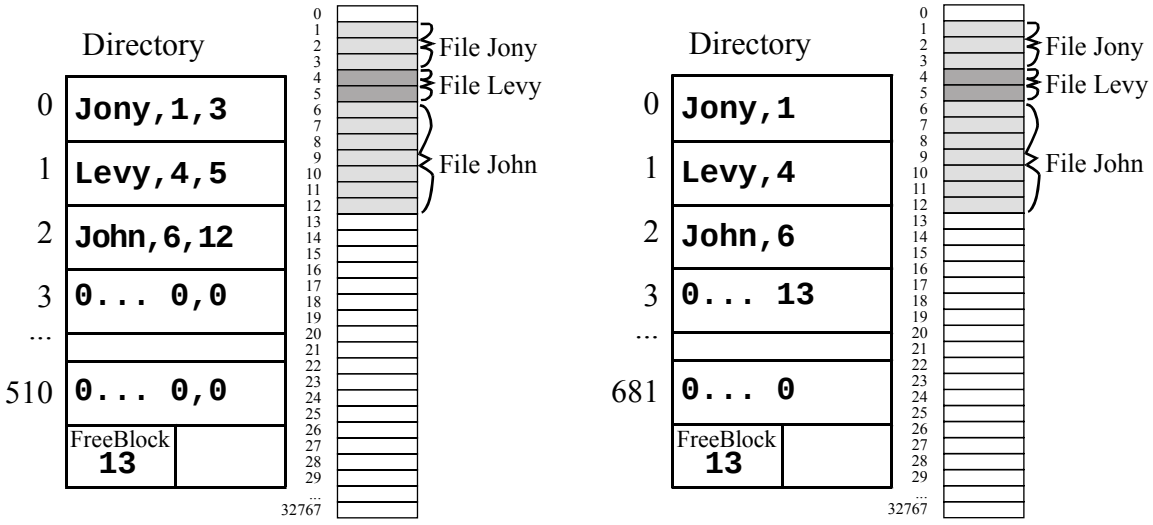
```
y =(3*x-2*y)/9
```

Solving for y/x, we get the DC gain

```
y/x = 3/11
```

(30) Question 4. In this system, files are *never deleted*.

Part a) See Spring 2009 Quiz 1 for a similar problem. 128 mebibytes is 2^{27} bytes, 4 kibibytes is 2^{12} bytes. So there are $2^{27}/2^{12} = 2^{15}$ blocks on the disk. Block 0 is the directory. Blocks 1 to 32767 are data blocks. Since data are never destroyed/deleted, all we need for free space is the block number of the first free block. If **`FreeBlock`** is 1, the disk is empty. If **`FreeBlock`** is 32676, it is full. In the picture **`FreeBlock`** is 13, meaning blocks 13 through 32767 are free.



Part b) The simplest way to organize data on this disk is *contiguous allocation*. For each file we need to know the first and last block. In this system, we never open an existing file and add more data to it.

Part c) We need 4 bytes for the name. If a name is less than 4 characters we can pad with null. The initial directory has nulls in the name field to signify empty directory entry. Each file also requires 2 bytes for the start block and 2 bytes for the last block of the file. This is a total of 8 bytes/file. We need 2 bytes for the **`FreeBlock`**, so 4094/8 rounded down is 511 files.

You can optimize by removing the 2 bytes for the last block of the file, because the first block of the next file will always be the last block plus one. This reduces the directory entry size to

6 bytes/file, so 4094/68 rounded down is 682 files. We will not be able to use the last directory entry for a file because it will not have a subsequent entry to store its size, so we can have 681 files.

Part d) At the end of the day you close the file. The last block written has a full 4096 bytes stored in it. In this application is there no internal fragmentation because the entire blocks are either used (have data) or free (have no data). The number of bytes stored in each file is an integer multiple of 4096.

(15) Question 5. The key is to not only detect errors, but correct them. Do a web search for RAID or Redundant Array of Independent Disks. Basically, you have 3 or more disks running in parallel. To make the probability of lost data less than or equal to p^2 we will use 3 disks. With three disks, we lose data if two disks fail on the same byte.

Part a) When you write a block, write it to all 3 disks. Of course you will want to implement the following program in such a way that the 3 disk writes operate in parallel, so the execution speed of 3 approximates the execution speed of one.

```
DRESULT nDisk_WriteBlock (const BYTE *buff, DWORD sector){
  return (eDisk_WriteBlock1(buff,sector) ||
          eDisk_WriteBlock2(buff,sector) ||
          eDisk_WriteBlock3(buff,sector));
}
```

Part b) When you read, you read from all three and return the majority **vote** of the three data buffers on a bit by bit fashion, a byte by byte fashion, or a block by block fashion. Again you will want to implement it in such a way that the 3 disk read operate in parallel, so the execution speed of 3 approximates the execution speed of one. You can mark any blocks that have disagreements as bad, first moving the data to a valid block then removing that bad block from the free space.

```
DRESULT nDisk_ReadBlock (BYTE *buff, DWORD sector){
  eDisk_ReadBlock1(buff1,sector);
  eDisk_ReadBlock2(buff2,sector);
  eDisk_ReadBlock3(buff2,sector);
  return vote(buff,buff1,buff2,buff3);
}
```

(bonus question) Question skipped. Consider a similar IIR digital filter. The sampling rate is 20 kHz, and the ADC is a 12-bit unsigned 0 to +3.3V range converter.

y(n) = 3x(n) – 2y(n-1)

Part a) Derive the H(z) transfer function for this filter.

Part b) Using H(z) calculate the gain of this filter at 10 kHz.