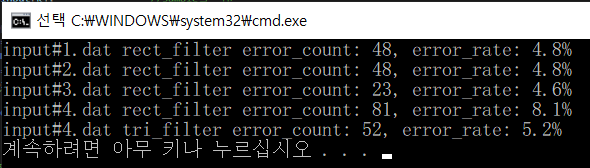
통신이론2 과제 #1

2016706018 박건희

각 데이터에 따른 비트에러 개수와, BER



- (1). input#1, input#2, input#3에 각각 matched filter를 사용하는 최적 수신기를 적용하여 BER를 구하고, 각 결과를 비교하고 성능 차이를 설명하시오.

Matched filter 설계시 input 1,2,3 같은 경우 모두 파형의 형태는 rect인 형태이기 때문에 matched filter를 rect인 형태로 matched filter를 설계할 수 있다. Rect의 크기보다는 파형이 중요하기 때문에 크기는1인 rect로 filter를 설계하였고 이는 즉 적분기의 형태이다. Bit의 판단은 가장 높은 SNR일 때 적합하기 때문에 bit당 sample의 마지막까지 적분 후 판단하였다.

신호가 노이즈에 의해 영향을 받았을 때, 판별시 위에서 설명한 것과 같이 적분시 그 1과 0의 경계선인 opt값을 정해주어야 한다. Input 1과 input 3은 bit ‘1’과 bit’0’이 진폭이 반대인 경우이기 때문에 0을 기준으로 설정시 최적의 필터가 될 것이고, input 2의 경우는 bit’0’일 때 0 bit ‘1’일 때 20이기 때문에 sample이 10개이므로 적분의 중간인 100을 opt값으로 최적의 필터로 설정하는게 좋다고 판단했다.

이를 통해 적분기를 bit당 sample수, bit수, opt값을 고려하여 함수를 만들어서 값을 도출한 결과

BER1=4.8%, BER2=4.8%, BER3=4.6%

위와 같은 BER이 도출되었다. 이때, input#2번 경우는 bit’1’의 에너지가 20\*20\*10인 4000이고 bit’0’의에너지는 0이다. 즉 input#2의 bit의 평균 에너지는 2000이다. Input#1과 input#3은 크기가 반대인 파형이기 때문에 bit당 에너지가 같다. input#3번의 경우는 bit당 sample 2배이고 즉 Tb­가 2배이다, 크기가 7.07배가 되었다. 동일한 노이즈 조건에서 즉 bit의 에너지는 7.07\*7.07\*20=999.698로 input#1의 10\*10\*10=1000과 거의 동일하다. 즉 Input#1과 Input#3은 거의 동일한 BER을 가지게 되고 위의 계산에서 약간의 오차 때문에 실제 프로그래밍을 통한 계산시 2개의 bit에러 차이가 났으나 이는 거의 무시할 수 있는 동일한 결과이다. Input#2의 경우 평균 bit에너지가 2배가 높아지기 때문에 input1이 에너지 효율이 input2보다 더 좋다.

-(2). input#4에 적분을 사용하는 수신기를 구현하여 BER를 구하시오. 다음, matched filter를 사용하는 최적 수신기를 적용하여 BER를 구하고, 이 결과와 적분을 사용하는 경우의 성능을 비교하시오. 또한, 최적 수신기의 성능을 input#1의 성능과 비교하시오

Input#4에서

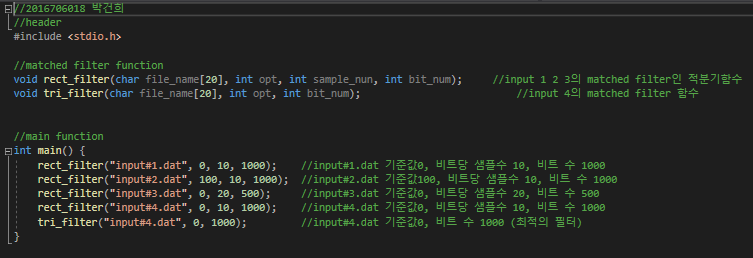
적분기를 이용한 경우 8.1%의 BER이 나게 되었다.

이 때, bit의 파형이 tri파형의 형태이기 때문에 matched filter를 동일한 모양으로 설계 후 위와 같이 코딩을 통해 확인한 결과 5.2%의 BER로 BER이 낮아졌다.

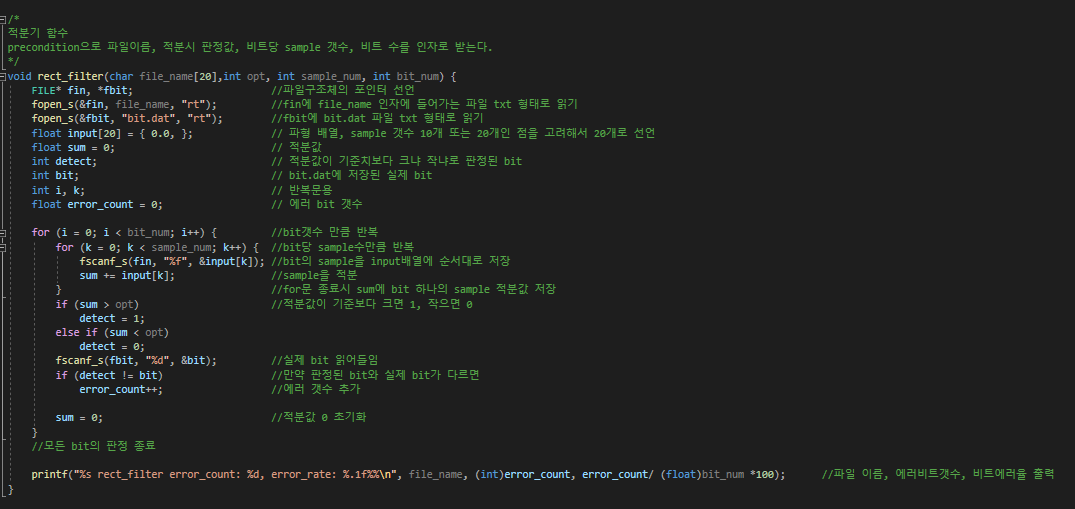
이를 통해 matched filter의 최적 성능은 파형이 동일할 때 나타나게 되는 것을 확인하였다.

Input#1과 Input#4에서 각 matched filter의 성능 비교시 각각 BER 4.8%와 5.2%로 Input#1에서 BER이 낮아서 성능이 조금 더 좋게 나타났다.

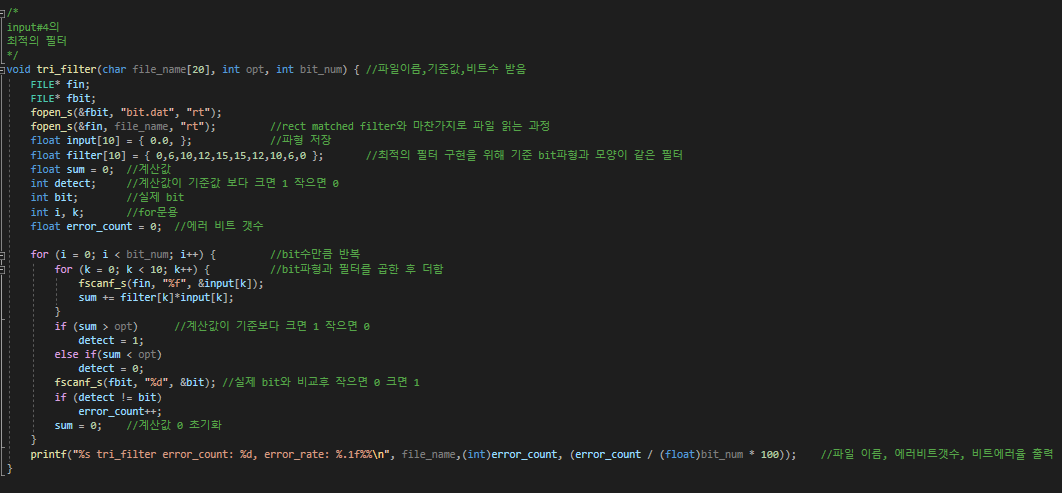
코드 설명



Main 함수에서 matched filter로 rect파형의 matched filter인 적분기 함수와 tri파형의 matched filter인 함수에 인자를 넣어서 동작하도록 구현했다.



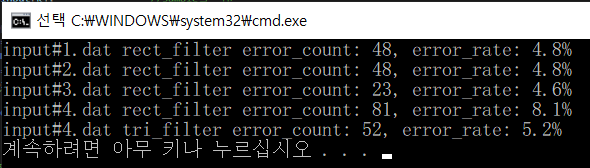
Rect\_filter 같은 경우는 input파일이름, opt값, bit당 sample갯수, 총 bit수를 인자로 받고  
sample 개수만큼 input파일을 읽고 전부 적분한 후 bit.dat에 들어있는 실제 bit와 비교하여 다를 때 error 개수가 올라가고, 출력에서 BER을 %단위로 나타내기 위해 error\_bit갯수/전체bit갯수\*100을 하였다.



Input#4의 matched filter인 tri\_filter 함수 같은 경우는 sample이 10개일 때 작동하므로 sample을 인자로 받는 precondition을 제외하고 구현하였다.

이 때 컨볼루션의 과정상 10개로 sample된 tri파형을 각각 곱한 후 더하여 opt에서 정해준 기준치와 비교 후 bit 판정을 한 후 bit.dat의 실제 bit와 비교하여 error bit의 개수를 측정하였다.

측정 후 BER을 %단위로 계산하였다.



출력 결과 차례로 input#1,2,3,4의 rect\_filter의 BER과 input#4의 tri\_filter의 BER이 위 사진과 같이 도출되었다.

C code 복사본

//2016706018 박건희

//header

#include <stdio.h>

//matched filter function

void rect\_filter(char file\_name[20], int opt, int sample\_nun, int bit\_num); //input 1 2 3의 matched filter인 적분기함수

void tri\_filter(char file\_name[20], int opt, int bit\_num); //input 4의 matched filter 함수

//main function

int main() {

rect\_filter("input#1.dat", 0, 10, 1000); //input#1.dat 기준값0, 비트당 샘플수 10, 비트 수 1000

rect\_filter("input#2.dat", 100, 10, 1000); //input#2.dat 기준값100, 비트당 샘플수 10, 비트 수 1000

rect\_filter("input#3.dat", 0, 20, 500); //input#3.dat 기준값0, 비트당 샘플수 20, 비트 수 500

rect\_filter("input#4.dat", 0, 10, 1000); //input#4.dat 기준값0, 비트당 샘플수 10, 비트 수 1000

tri\_filter("input#4.dat", 0, 1000); //input#4.dat 기준값0, 비트 수 1000 (최적의 필터)

}

/\*

적분기 함수

precondition으로 파일이름, 적분시 판정값, 비트당 sample 갯수, 비트 수를 인자로 받는다.

\*/

void rect\_filter(char file\_name[20],int opt, int sample\_num, int bit\_num) {

FILE\* fin, \*fbit; //파일구조체의 포인터 선언

fopen\_s(&fin, file\_name, "rt"); //fin에 file\_name 인자에 들어가는 파일 txt 형태로 읽기

fopen\_s(&fbit, "bit.dat", "rt"); //fbit에 bit.dat 파일 txt 형태로 읽기

float input[20] = { 0.0, }; // 파형 배열, sample 갯수 10개 또는 20개인 점을 고려해서 20개로 선언

float sum = 0; // 적분값

int detect; // 적분값이 기준치보다 크냐 작냐로 판정된 bit

int bit; // bit.dat에 저장된 실제 bit

int i, k; // 반복문용

float error\_count = 0; // 에러 bit 갯수

for (i = 0; i < bit\_num; i++) { //bit갯수 만큼 반복

for (k = 0; k < sample\_num; k++) { //bit당 sample수만큼 반복

fscanf\_s(fin, "%f", &input[k]); //bit의 sample을 input배열에 순서대로 저장

sum += input[k]; //sample을 적분

} //for문 종료시 sum에 bit 하나의 sample 적분값 저장

if (sum > opt) //적분값이 기준보다 크면 1, 작으면 0

detect = 1;

else if (sum < opt)

detect = 0;

fscanf\_s(fbit, "%d", &bit); //실제 bit 읽어들임

if (detect != bit) //만약 판정된 bit와 실제 bit가 다르면

error\_count++; //에러 갯수 추가

sum = 0; //적분값 0 초기화

}

//모든 bit의 판정 종료

printf("%s rect\_filter error\_count: %d, error\_rate: %.1f%%\n", file\_name, (int)error\_count, error\_count/ (float)bit\_num \*100); //파일 이름, 에러비트갯수, 비트에러율 출력

}

/\*

input#4의

최적의 필터

\*/

void tri\_filter(char file\_name[20], int opt, int bit\_num) { //파일이름,기준값,비트수 받음

FILE\* fin;

FILE\* fbit;

fopen\_s(&fbit, "bit.dat", "rt");

fopen\_s(&fin, file\_name, "rt"); //rect matched filter와 마찬가지로 파일 읽는 과정

float input[10] = { 0.0, }; //파형 저장

float filter[10] = { 0,6,10,12,15,15,12,10,6,0 }; //최적의 필터 구현을 위해 기준 bit파형과 모양이 같은 필터

float sum = 0; //계산값

int detect; //계산값이 기준값 보다 크면 1 작으면 0

int bit; //실제 bit

int i, k; //for문용

float error\_count = 0; //에러 비트 갯수

for (i = 0; i < bit\_num; i++) { //bit수만큼 반복

for (k = 0; k < 10; k++) { //bit파형과 필터를 곱한 후 더함

fscanf\_s(fin, "%f", &input[k]);

sum += filter[k]\*input[k];

}

if (sum > opt) //계산값이 기준보다 크면 1 작으면 0

detect = 1;

else if(sum < opt)

detect = 0;

fscanf\_s(fbit, "%d", &bit); //실제 bit와 비교후 작으면 0 크면 1

if (detect != bit)

error\_count++;

sum = 0; //계산값 0 초기화

}

printf("%s tri\_filter error\_count: %d, error\_rate: %.1f%%\n", file\_name,(int)error\_count, (error\_count / (float)bit\_num \* 100)); //파일 이름, 에러비트갯수, 비트에러율 출력

}