

Analog and Digital Techniques for Fast Ultrasonic Processing

Matthew Aldrich
Yale University

April 26, 2004

Abstract

An analog circuit capable of ultrasound detection has been designed and simulated. At 8m, the circuit yields a 100mV peak to peak output. A detection algorithm using correlation techniques has been designed and simulated. The algorithm behaves as predicted from MATLAB simulations.

1 Introduction

Ultrasound ad-hoc sensor networks have been designed and realized. Successful operation of these networks relies upon accurate time of flight (TOF) measurements. Accurate TOF measurement can be used to determine the distance between a transmitter and receiver, or in some cases, the location of an object. Current methods proposed by Savvides et al incorporate a threshold detection method to determine the TOF. However, it can be shown that a simple threshold detection circuit is quite sensitive to noise and in some cases, yields TOFs that appear longer than they truly are [1].

This paper discusses two methods of ultrasonic detection. One describes in detail the analog detection circuit and the other discusses an algorithm which correlates the received signal with one in memory.

2 Analog Detection

Savvides et al, the group responsible for the AHLoS system, realized an ultrasonic platform that used threshold detection to determine an incoming ultrasonic "chirp." [2] This process is implemented through analog circuitry. An incoming chirp is amplified through non inverting amplifiers and then fed through a comparator set at a reference voltage. When the incoming signal exceeds the reference voltage, a voltage pulse, signifying the correct identification of the ultrasound pulse, is transmitted. Savvides' schematic shall serve as the basis for our implementation of this analog technique.

2.1 Circuit Basics

Assuming the reader is familiar with basic circuit theory, the operation of this circuit is quite straightforward and simple. In this description, the reference numbers refer to the circuit schematic. The circuit is broken down as follows:

- AC coupling and decoupling of signal
- Setting a DC operating point
- Gain stages
- Threshold Detection

2.1.1 AC Coupled Signals

Electrolytic capacitors C1 and C2 ensure that the DC characteristics of the transmitted signal are blocked and that the DC voltage from the power supply is added.

2.1.2 Setting a DC Operating Point

Resistors R1 and R2 set the DC operating point in this circuit. By connecting R3 to this voltage an offset in the first gain stage is created. The technique used in R1 and R2 is the same as in R8 and R9. This voltage divider equation is given as follows (using R1 and R2):

$$V_{out} = V_{in} \frac{R_2}{R_2 + R_1} \quad (1)$$

2.1.3 Gain Stages

UA1 and UA2 are the input stages found in LM358 operational amplifier. In recreating this circuit, any low noise, single supply amp may be used. V_{ss} in this case is ground. Resistors {R3,R4} and {R6,R7} control the gain in this circuit, in the form of a non-inverting configuration. Capacitors {C4,C6} set the roll off frequency (essentially filtering out noise). The gain of each stage may be calculated as follows:

$$G = \frac{V_{out}}{V_{in}} = \left(1 + \frac{R_2}{R_1}\right) \quad (2)$$

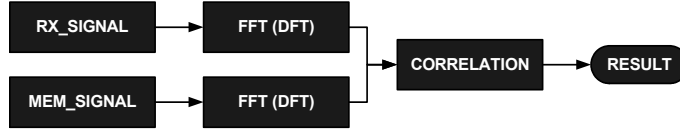
In this circuit, we have closed loop gains of roughly 90 and 600. If better performance is desired, the gain stages can be increased, and the step up ratio decreased.

2.1.4 Threshold Detection

The output of U1B pin 7 is divider according the ratio of R8 and R9. This voltage will be "sitting" on the DC reference voltage at pin 3 of U2A. At pin 2 of U2A, the voltage set by resistors R1 and R2 will serve as the threshold.

3 Detection Algorithm

As an alternative to using discrete components, the on board processor allows for more complex signal processing. The algorithm consists of this central idea: Sample the signal, convert the signal to the frequency domain, convert a similar signal in memory to the frequency domain, perform the correlation of the two signals. This can be summarized by the following:



High level description of algorithm

the rest of this section will break down the individual elements of this algorithm and describe their operation.

3.1 The Frequency Domain and the DFT

Once the input is sampled the first step is to convert the signal into the frequency domain. Assuming some signal analysis background, only the information essential to algorithm development is given. The discrete Fourier transform is given as:

$$X_k = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N} \tag{3}$$

where the rectangular form is:

$$X_k = R_k + jI_k \tag{4}$$

and:

$$R_k = x(0) + \sum_{n=1}^{N-1} x[n] \cos \frac{2\pi kn}{N} \tag{5}$$

$$I_k = - \sum_{n=1}^{N-1} x[n] \sin \frac{2\pi kn}{N} \tag{6}$$

and the inverse DFT is:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{j2\pi kn/N} \tag{7}$$

These equations are used to compute the DFT and IDFT of input data in the c program.

3.2 Convolution and Correlation

The convolution of a signal is simply the sum of one signals response to another being slid through it. The correlation is a cousin of convolution. If the reference is a close copy of the received signal and lags for some t, the correlation will be a positive value; if it leads, the correlation will produce large negative values.

Convolution in the frequency domain is given as:

$$x[n] * v[n] \Leftrightarrow X(\omega)V(\omega) \tag{8}$$

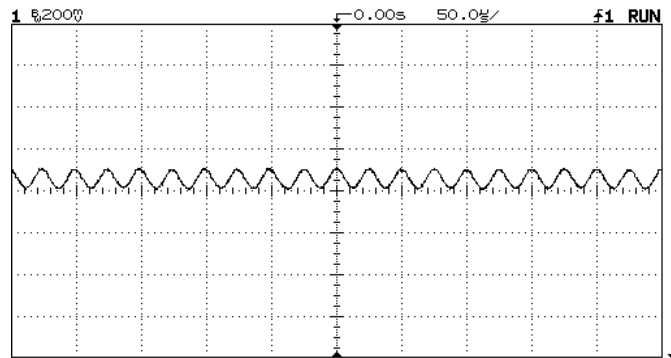
and the correlation by:

$$x[n] * v[n] \Leftrightarrow X(\omega)V(\omega)^* \tag{9}$$

where * denotes the complex conjugate. Because these operations are cyclical, end effects must be taken into consideration. To avoid spoiling the correlation data, the sampled signal should be zero padded after the last collected data point. An example is given in the project presentation file and in [3].

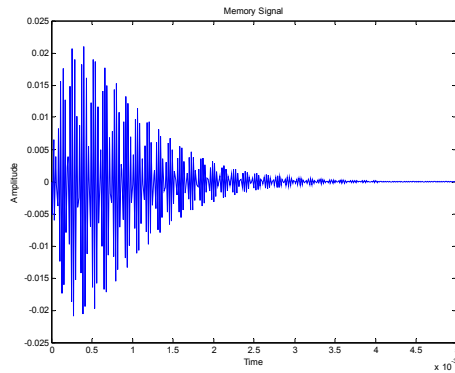
4 Results

The analog circuit was prototyped and verified. Using a supply voltage of 3.3V, a continuous input waveform of 3.3V at 40KHz, and a worst case distance of 8m, we achieve a detection waveform of 100mV peak to peak. The amplitude can be made greater by increasing the gain stages.

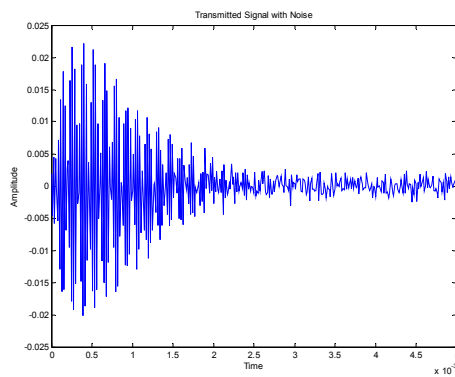


Oscilloscope reading of test case

The algorithm has been coded in C and can be found in the appendix. Because the board was unable to be initialized, the ADC was not implemented. However, the program does verify the correctness of the algorithm and code. The algorithm was tested with the following data (generated in MATLAB) using the equations listed in [4]:

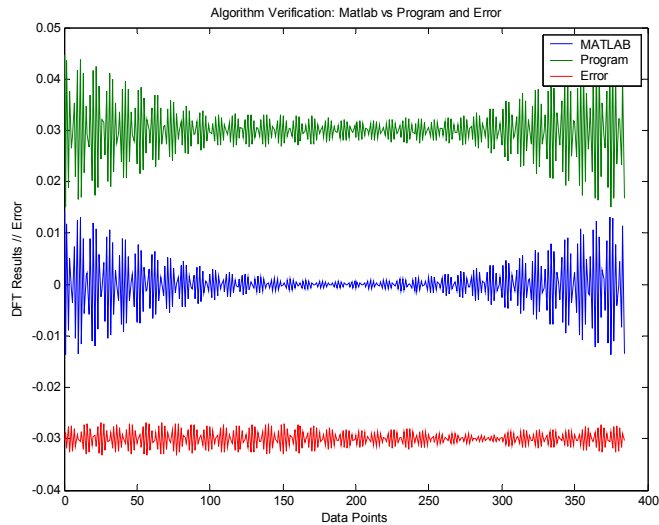


Memory Reference Signal



Received Signal

The following is how the algorithm compares to MATLAB's FFT and IFFT (data has .03 offsets for readability).



Output of algorithm, MATLAB, and error between signals.

where the error between the operations is practically negligible thus verifying the correctness of the algorithm.

5 Future Work

This project is merely a good starting point in the design of robust detection methods. On the algorithm side much work still needs to be done. In particular, the data may need to be zero padded so that the correlation data does not peak at zero. In addition, a threshold value must be selected at the earliest possible sample point so that a signal can be detected.

Furthermore, if the system needs to perform the correlation among a variety of signals, an encoding scheme must be selected. Presently, pulse-width modulation (PWM) appears to be the best option. In addition, tables would need to be created in the program to allow for lookup. For these reasons, analog detection is simple and in most cases adequate.

6 Conclusions

This paper describes the analog techniques necessary for proper ultrasound detection. In addition, it discusses and documents the algorithm and code necessary for digital detection. The analog simulations provide promising results at large distances and the algorithm is in accordance with MATLAB data. In

addition, this paper describes the next steps towards a fully functional analog/digital solution.

References:

- [1] Barshan, B, "Fast Processing Techniques for Accurate Ultrasonic Range Measurements," Meas. Sci Tech. 11 (2000) 45-50
- [2] Savvides, A et al, AHLoS system, <http://nesl.ee.ucla.edu/projects/ahlos/>
- [3] Press, W. H., Trukolsky, S. A., Vetterling, W. T., and Flannery, B. T. (1992). Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press
- [4] Parrilla, M et al, "Digital Signal Processing Techniques for High Accuracy Ultrasonic Range Measurements" IEEE Transactions on Instrumentation and Measurement/ 40 (1991) 759-763

A Algorithm Code

data sets are provided at course webpage <http://pantheon.yale.edu/~mha4/ee449/>
special thanks to Michael Bell whose assistance in writing this program allowed for speedy completion.

B Main.C

```
#include <stdio.h>
#include "dft.h"

struct data {
    double time;
    double val;
};

struct sig {
    int n;          //number of points in the sample
    struct data *d; //data at those points [ (time,value) pairs ]
    double *v;
};

void print_sig(struct sig *s) {
    int i;

    for(i=0; i<s->n; i++) {
        printf("(%1.10e, %1.10e)\n", s->d[i].time, s->d[i].val);
    }
}

int read_into(char *filename, struct sig *s) {
    FILE *fid;
    int i;

    fid = fopen(filename, "r");

    //printf("opening %s for read\n", filename);

    fscanf(fid, "%d\n", &(s->n));
    //printf("s->n %d \n", s->n);
    s->d = (struct data *) malloc(s->n * sizeof(struct data));
    s->v = (double *) malloc(s->n * sizeof(double));
    for(i=0; i < s->n; i++) {
        fscanf(fid, "%lf,%lf\n", &(s->d[i].time), &(s->d[i].val));
        s->v[i] = s->d[i].val;
    }
}
```

```

}

fclose(fid);
//with girls, you're never sure what they're going to turn off, like
// they might turn just the computer off and leave the monitor and
// the printer on.  even better - they just turn the monitor off.

return 0; //TODO(?) return nonzero on error.
}

double *cplx_mult(double *a, double *b, int n) {
    int i;
    double *ret;

    ret = (double *) malloc(2 * n * sizeof(double));
    for(i = 0; i < n; i++) {
        ret[2 * i] = a[2 * i] * b[2 * i] + a[2 * i + 1] * b[2 * i + 1];
        ret[2 * i + 1] = - a[2 * i] * b[2 * i + 1] + a[2 * i + 1] * b[2 * i];
        /* ret[2*i] = ret[2*i]/n;
        ret[2*i+1] = ret[2*i+1]/n;
        */
    }

    return ret;
}

int main(int argc, char **argv) {
    struct sig mem; //signal in memory
    struct sig sample; //signal being sampled, tested against all sigs in mem
    double *dft_mem, *dft_sam;
    double *corr, *conv;
    int i;

    //printf("argc: %d\n", argc);
    if (argc != 3) {
        printf("usage: %s <mem_file> <signal_file>\n", argv[0]);
        exit(1);
    }

    read_into(argv[1], &mem);
    read_into(argv[2], &sample);

    // Looks good
    /*
    printf("memory:\n\n");

```

```

print_sig(&mem);

printf("\n\nsampled signal:\n\n");
print_sig(&sample);
*/

//Guy, makefile all the time don't work

dft_mem = dft(mem.v, mem.n);

/*
printf("dft of our mem signal:\n");
for(i = 0; i < mem.n; i++) {
    printf("%d: %f + %f i\n", i, dft_mem[2 * i], dft_mem[2 * i + 1]);
}
*/

dft_sam = dft(sample.v, sample.n);
/*
printf("dft of our sampled signal:\n");
for(i = 0; i < sample.n; i++) {
    printf("%d: %f + %f i\n", i, dft_sam[2 * i], dft_sam[2 * i + 1]);
}
*/

conv = cplx_mult(dft_mem, dft_sam, mem.n);
/*
printf("convolution of our signals:\n");
for(i = 0; i < mem.n; i++) {
    printf("%d: %f + %f i\n", i, conv[2 * i], conv[2 * i + 1]);
}
*/
corr = inv_dft(conv, mem.n);
printf("correlation of our signals:\n");
for(i = 0; i < mem.n; i++) {
    printf("% 1.10e\n", corr[i]);
}
}

```

C DFT.C

```
#include <math.h>
/*
  Discrete Fourier Transform
  naive (definitely the way we're going to do it)

  n      number of sampled points
  data is the real input

*/
double *dft(double *data, int n) {
  double *tmp;
  int i,j;

  tmp = (double *) malloc( sizeof(double) * n * 2 );

  for(i = 0; i < n; i++) {
    tmp[2 * i] = tmp[2*i+1] = 0;  //zero the values

    tmp[2 * i] += data[0];
    for(j = 1; j < n; j++) {
      tmp[2 * i] += data[j] * cos(2 * M_PI * i * j / n);    //real
      tmp[2 * i + 1] -= data[j] * sin(2 * M_PI * i * j / n); //imag
    }
  }
  return tmp;
}

double *inv_dft(double *data, int n) {
  double *tmp;
  int i,j;

  tmp = (double *) malloc( sizeof(double) * n );

  for(i = 0; i < n; i++) {
    tmp[i] = 0; //it's all right.  you're initializing it. ;)

    for(j = 0; j < n; j++) {
      tmp[i] += data[2 * j] * cos(2 * M_PI * i * j / n);    //real
      tmp[i] -= data[2 * j + 1] * sin(2 * M_PI * i * j / n); //imag
    }
    tmp[i] = tmp[i] / n;
  }
  return tmp;
}
```

D DFT.h

```
double *dft(double *data, int n);  
double *inv_dft(double *data, int n);
```

E Makefile

```
correlate: main.c dft.c dft.h  
\quad gcc -o correlate main.c dft.c -lm
```

```
clean:  
\quad rm -f *~correlate
```