Simple Analog Sensor Filtering for Botball

David Culp
DeWitt Perry Middle School (culpd@cfbisd.edu)

Simple Analog Sensor Filtering for Botball

I. Introduction

Sensors are rarely 100% reliable and will typically have some amount of noise in the sensor readings. By noise we mean unwanted signals in the sensor reading that come from a source other than what we are monitoring This noise can come from a variety of places that include but are not limited to:

- The electronics inside the sensor itself.
- The electronics of the microcontroller.
- The batteries and other power sources.
- Motors and servos.
- Outside electrical *interference*.
- Transient effects in the environment.

There are entire branches of electronics and computer science devoted solely to reducing or eliminating noise. These techniques can get extremely mathematical and technical and are overkill for typical Botball usage.

II. Typical Analog Sensor Usage in Botball

A typical Botball program will tend to key off of a single value and a single sensor reading, an example of this technique appears in the code listing below:

Listing 1

```
while(analog10(some_sensor_port)<500)
{
     do_some_stuff();
}
do_some_other_stuff();
```

This tends to work fairly well in most circumstances in Botball. However, if the lighting changes, or the sensor passes over small unforeseen marks on the table, or the threshold value is very close to the signal value then the sensor could trigger early or late. At DeWitt Perry Middle School, because of years of reuse, our tables will often have stray marks on them left from previous years that are impossible to remove entirely, these stray marks can interfere with sensor readings. In addition, the Botball IR distance sensor (“ET” Sensors) and sonar sensors are particularly noisy and can give rapidly changing values, even when held “steady”.

III. Averaging Sensor Values

Averaging sensor values is one way to deal with noise and the most common way my students come up with to solve the problem. The technique is pretty simple, we simply read the sensor a set number of times, keeping track of the total and then dividing that result by the number of readings. A typical function to carry this out appears below:

Listing 2

```
int analog_average(int port,int num_samples)
{
    int count, total;
    total=0;
    for(count =0; count<num_samples;count ++)
    {
        total+=analog10(port);
    }
    return total/num_samples;
}
```

Taking the average of a number of samples has some advantages: It is fast and not very taxing on the processor, it is easy to implement and it works reasonably well. However, averaging a number of samples may still include values well outside of the range of expected values in our calculations and skew the results. In an environment in which the expected value is very close to the noise or a transient event occurs for a significant portion of the sample time then this technique may not work very well. In fact, it may perform no better than taking a single reading in some cases.

IV. Taking the Median Value

As discussed briefly, taking the average of a number of samples may not work well as you are possibly including values outside of the expected value in the calculations, thereby skewing the overall result. A better technique is to take a number of readings, sort those readings from low to high and take the median value (the value in the middle). This will help to filter out both spikes and dips in the sensor readings as the high spikes and low dips will occur at the extremes of the data range. A typical function to do this appears in listing 3.

Listing 3

```
int analog_median(int port, int num_samples)
{
      int *array, count, return_val;
      array = (int *)malloc(sizeof(int)*num_samples);

      for(count=0; count<num_samples; count++)
      {
            array[count]=analog10(port);
      }

      sort_array(array, num_samples);
      return_val=array[num_samples/2];
      free(array);
      return return_val;
}
int sort_array(int *array, int size)
{
      //Inplements a simple bubble sort
      int n,k, is_sorted, temp;
      k=0;
      //Sort the array
      do
      {
            is_sorted=TRUE;
            for(n=0; n<(size-k-1);n++)
            {
                  if (array[n] > array[n+1])
                  {
                        is_sorted=FALSE;
                        temp = array[n+1];
                        array[n+1]=array[n];
                        array[n]=temp;
                  }
            }
            k++;
      }while(is_sorted==FALSE);

      return 0;
}
```

The function “analog_median” takes the port number the sensor is plugged into and the number of samples to take and returns the value found in the middle of the array. The function is dependent on another function; “sort_array” which takes an array of integers and the number of values in the array and sorts them from lowest to highest. This technique works extremely well and is probably all you need for Botball.

V. Combining the Two Techniques

Taking the median value of a number of readings works extremely well, does not tax the processor too much and is probably all you will ever need in Botball. However, you can combine the above two techniques by taking a number of readings, sorting them into order and then taking the average of the readings in the middle. Under some circumstances this may work better than simply taking the average or median value. Listing 4 shows a function to accomplish this task:

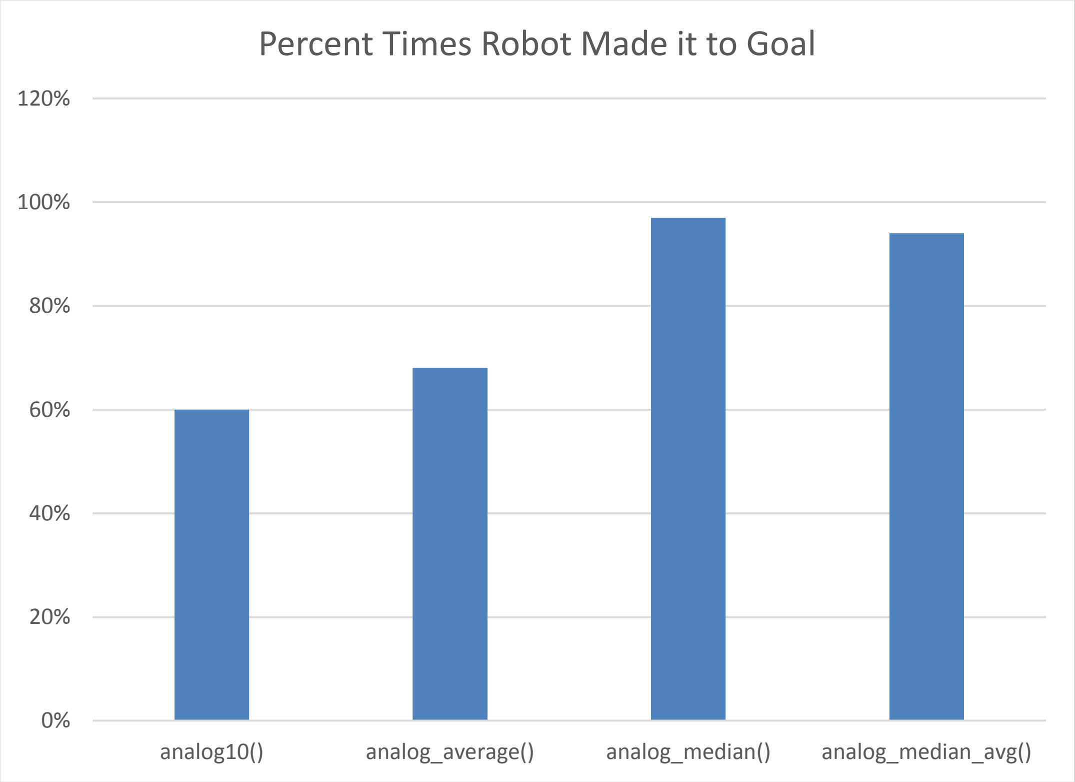
Listing 4

```
int analog_median_avg(int port, int num_samples)
{
        int    *array, count, total, return_val;
        total=0;
        array = (int *)malloc(sizeof(int)*num_samples);

        for(count=0; count<num_samples; count++)
        {
                array[count]=analog10(port);
        }

        sort_array(array, num_samples);

        for(count = (num_samples/2)-5; count < (num_samples/2)+5; count ++)
        {
                total+=array[count];
        }
        return_val = total/10;
        free(array);
        return return_val;
}
```

VI. Testing the Filtering Functions

In order to test the techniques and show that they can work better than simply keying off of a single reading a simple robot was built with a single IR reflectance sensor at the front. The robot was set on a typical Botball table field with a strip of black duct tape 6 feet away. The goal was to have the robot drive forward until it reached the black duct tape. Values were obtained for the sensor reading over white and over the black tape. A number of runs were then done with a “clean” table and then a number of runs were done in which small black marks were left on the table and different “shadow” zones were created with varying light intensities. The random black marks were meant to simulate parts that may have fallen off on the table, scuff marks left by wheels and stray marks that may have been made by pencils or markers. For the analog filtering functions, 50 sample readings were taken each function call.

In the case of a 100% clean table, the filtering functions performed no better than simply taking a single reading, the robot made it to the black tape 100% of the time in all tests. However, as I increasingly placed anomalous marks and debris in the way, the robot using the filtering routines was able to make it to the black tape almost 100% of the time while the robot using only a single reading triggered its threshold early and stopped short of the black tape almost 40% of the time.



VII. Caveats and Extending the Technique beyond Analog Sensors

The KIPR LINK is not extremely fast and does not have a real-time operating system; therefore the user should keep the number of samples as low as possible. In other words, do not try to read and sort 1,000 samples each function call in real time. In general 10-50 samples work well and are a good tradeoff between speed and accuracy. Although all of the techniques work, choosing which to use will depend on the exact circumstances. In the vast majority of cases the "analog_median" function with around 50 values will work the best.

The above techniques work extremely well for the analog sensors included in the Botball kit. However, the techniques can be extended to filter data coming from the camera and the SONAR sensor.

Complete Listing

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

#define TRUE 1
#define FALSE 0

int sort_array(int *array, int size);
int analog_average(int port,int num_samples);
int analog_median(int port, int num_samples);
int analog_median_avg(int port, int num_samples);

int main(void)
{

      while(1)
      {
            printf("R: %d AA:%d AM:%d AMA: %d\n",
analog10(1),analog_average(1, 50), analog_median(1,50),
analog_median_avg(1,50) );
            msleep(750);
      }

}

/*
Function Name: analog_average
Purpose: returns the average of num_samples readings from analog port
Argguments:
      int port - the analog port to read from
      int num_samples - the number of samples to average
*/
int analog_average(int port,int num_samples)
{
      int count, total;
      total=0;
      for(count =0; count<num_samples;count ++)
      {
            total+=analog10(port);
```

```
	}
	return total/num_samples;
}

/*
Function Name: analog_median
Purpose: returns the medain value of num_samples reading from port numer port
	int port - the analog port to read from
	int num_samples - the number of samples to average
*/
int analog_median(int port, int num_samples)
{
	int *array, count, return_val;
	array = (int *)malloc(sizeof(int)*num_samples);

	for(count=0; count<num_samples; count++)
	{
		array[count]=analog10(port);
	}

	sort_array(array, num_samples);

	return_val=array[num_samples/2];
	free(array);
	return return_val;
}

/*
Function Name: analog_median_avg
Purpose: returns the average of the 10 medain values of num_samples reading
from port numer port
	int port - the analog port to read from
	int num_samples - the number of samples to average

Notes: num_samples MUST be greater than 10
*/
int analog_median_avg(int port, int num_samples)
{
	int	*array, count, total, return_val;
	total=0;
	array = (int *)malloc(sizeof(int)*num_samples);

	for(count=0; count<num_samples; count++)
	{
		array[count]=analog10(port);
	}

	sort_array(array, num_samples);
	for(count = (num_samples/2)-5; count < (num_samples/2)+5; count ++)
	{
		total+=array[count];
	}
	return_val = total/10;
	free(array);
	return return_val;
}

int sort_array(int *array, int size)
{
	//Inplements a simple bubble sort
	int n,k, is_sorted, temp;
	k=0;
	//Sort the array
	do
	{
		is_sorted=TRUE;
		for(n=0; n<(size-k-1);n++)
		{
			if (array[n] > array[n+1])
			{
				is_sorted=FALSE;
				temp = array[n+1];
				array[n+1]=array[n];
				array[n]=temp;
			}
		}
		k++;
	}while(is_sorted==FALSE);

	return 0;
}
```