RangeTrack: Object Detection, Object Identification, and Robot Localization with the Sharp "ET" Rangefinder (Part 1)

Jeremy Rand and Emily Curtis
Norman Advanced Robotics
jeremy.rand@ou.edu, coconut231@yahoo.com

RangeTrack: Object Detection, Object Identification, and Robot Localization with the Sharp "ET" Rangefinder (Part 1)

1 Introduction

Ever since the CMUCam was introduced to Botball in 2003, Botball teams have been trying to use cameras to identify and locate objects and to correct the robot's position. Unfortunately, most teams have met with limited success. Even the top-of-the-line XBCCam yielded unstable results in many cases. Subtle lighting changes can completely throw off the color detection algorithms, and even a single frame of bad tracking data can confuse robots which aren't expecting it. For many Botball teams who attempt to utilize the camera, practice tables serve the primary purpose of calibrating the camera models, reducing time for other test runs and debugging. The CBCCam introduced additional problems due to its high-latency operation (tests conducted for *Hacking the CBC Botball Controller* [1] yielded an average lag time of about 1/3 of a second). Due to the prohibitively high amount of code necessary to make the camera sufficiently reliable for tournament play, a better system is desirable.

If the camera is not the solution, what is? Many teams have used the Sharp IR Rangefinder (often called "ET" in Botball circles) for simple object detection, but this also yields problems. The big issue is that the ET doesn't detect size, shape, or angular position; only distance. However, since the ET updates quickly (every 39ms) and has a narrow beam width, multiple ET readings with slightly differing aim can yield useful data.

Why would you want to use an ET instead of a camera for tracking objects? Because of the camera's lag of 1/3 second, if your robot is turning until it's centered on an object, your robot either will have to turn very slowly or will overshoot and oscillate before finding the object. In contrast, the ET's 39ms update rate means that if you want to take a measurement every 5 degrees for a 50-degree range (quite possibly overkill in Botball), you can find the object in a worst-case elapsed time of ~0.4 seconds, compared to ~3.3 seconds with a webcam. In addition, rangefinders can easily see objects such as the transparent cups from the Botball 2009 season, or objects which are in poor lighting such as in the Beyond Botball 2008 cave, while a typical webcam will fail to identify these objects.

The ET is also better suited for localization (estimating your robot's position based on sensor readings) than the camera, because every millimeter of every PVC wall on the game board is now a data point, rather than a single colored target. More data points give more precision, with

random noise having less impact, and PVC walls are available in every part of the game board regardless of which way your robot is facing.

Another advantage of the ET is that camera vision is a very computationally expensive process. The vision system on the CBC will easily eat 50% of your CPU, and the XBC had to leave most of the work to expensive hardware external to the CPU (the Field-Programmable Gate Array). Object tracking with the ET is very fast, and can even be done by an XBC using only the GBA's 16MHz processor -- making the 350MHz Chumby more than capable of doing so without taxing the system. Finally, this research was started when the Botball kit included 2 webcams (only 1 per CBC) but 3 ET's (all of which can be on a single CBC), which made rangefinder object tracking attractive to teams who don't think 1 webcam per robot is enough. (That particular reason is now obsolete since 2 cameras can be used in a single CBC [2], and the Botball kit now only has 1 ET.)

2 Basic Design Components

RangeTrack consists of four major steps: logging, segmentation, object analysis, and localization. Logging consists of reading position and range data, resulting in a set of two-dimensional points where the rangefinder detected something. Segmenting consists of identifying which subsets of the logged points correspond to which physical objects. Object analysis consists of applying statistical tests to the segmented objects to identify them, e.g. as a pom stack or a wall. Finally, localization uses applies additional statistical tests to the objects detected as walls to estimate the robot's position. All of these components must work reliably for RangeTrack to function, except localization, which is optional. RangeTrack gives each of these components its own C++ class. Some helper classes are also present, e.g. a logged data anti-noise filter. RangeTrack is intended to be extensible, so a user can add support for custom sensors, noise filters, and segmentation algorithms, or even entirely new analysis capabilities. In addition to the library which runs on the robot, RangeTrack includes the capability to export data to a PC for further analysis and simulation.

Because RangeTrack is written in C++, it requires the NHS Patchset firmware with Code::Blocks [3] [6]. As with all unofficial modifications to the CBC firmware, the KIPR warranty does not cover damage caused by the NHS Patchset or RangeTrack, and although we don't believe such damage is likely, we are unable to provide a warranty ourselves. As such, Botballers should be aware of what they're getting into before using RangeTrack.

3 Installation

First off, download RangeTrack [4] and the NHS Patchset Firmware [6]. RangeTrack uses the Shared Libraries feature of the NHS Patchset firmware. Instructions for using Shared Libraries are in CBC Hacking 2010 [3]. Once you've loaded the libRangeTrack project onto your NHS Patchset-hacked CBC, just create a new C++ CBC project in Code::Blocks, and add RangeTrack to the library list. Detailed instructions for all of these procedures are in CBC Hacking 2010 [3]. You're now ready to code with RangeTrack!

4 Basic Usage

To get started, try the following code:

```
#include "RangeTrackAnalyzer.h"
#include "RangeTrackCreateDistancePositionReader.h"
#include "RangeTrackConstantPositionReader.h"
#include "RangeTrackETRangeReader.h"
#include "RangeTrackMedian5Filter.h"
#include "RangeTrackMeanFilter.h"
#include "RangeTrackVectorFilter.h"

#define ETPORT 6

RangeTrackAnalyzer *rt;
vector<RangeTrackFilter *> filterlist;

int main()
{
    // Port 6 is floating
    set_each_analog_state(0, 0, 0, 0, 0, 0, 1, 0);

    // Connect to Create
    create_connect();
    printf("Connected to Create\n");

    // Setup filters (Median+Mean, both with diameter 5)
    filterlist.push_back(new RangeTrackMedian5Filter());
    filterlist.push_back(new RangeTrackMeanFilter(5));

    // Create new Analyzer
    rt = new RangeTrackAnalyzer(
    new RangeTrackCreateDistancePositionReader(), // Create Distance is X
axis
    new RangeTrackConstantPositionReader(0), // Y axis is always 0
    new RangeTrackConstantPositionReader(90 * 3142 / 180),
        // Theta orientation is always 90 degrees, converted to milliradians
    new RangeTrackETRangeReader(ETPORT), // Rangefinder is an ET
    new RangeTrackVectorFilter(filterlist), // All filters in filterlist
    new RangeTrackFilter(), // Ignore this null filter, it's just boilerplate
    new RangeTrackFilter() // Ignore this null filter, it's just boilerplate
       );
    printf("Created rt\n");

    gc_distance = 0;

    printf("Initializing segmentation parameters\n");
```

```
    rt->SetMaxRange(600); // Ignore objects >600mm away

    // Edges are within 150mrad of the sensor beam;
    // keep these values at the same magnitude unless
    // you know what you're doing!
    rt->SetMinAngle(-150);
    rt->SetMaxAngle(150);

    printf("Sweeping 1 meter\n");

    create_drive_straight(100);

    while(gc_distance < 1000)
    {
        // Update odometry
        create_distance();

        // Update RangeTrack
        rt->UpdateAll();

        // Don't hog all the CPU
        msleep(5);
    }

    create_stop();

    printf("Done sweeping; dumping\n");

    // Dump data
    rt->DumpLog("/tmp/rt_log_straight.csv");
    rt->DumpSegment("/tmp/rt_segment_straight.csv");

    printf("%d segments found.", rt->GetNumSegments());

    printf("Dump complete; exiting\n");
}
```

This code will drive a Create 1 meter, and report any objects found by an ET aimed perpendicular to the Create's path. What's it doing under the hood? Read on.

5 Logging

RangeTrack logs data using sensor drivers. A sensor driver is simply a C++ class which implements a method to read it. Both odometry sensors (for measuring the position of the sensor) and range sensors (for measuring the distance between the sensor and the object) are available. The sensors which are currently implemented are:

- Odometry

- Create Distance
  - `new RangeTrackCreateDistancePositionReader()`
  - Returns the distance from the iRobot Create chassis in millimeters.
- Create Angle
  - `new RangeTrackCreateAnglePositionReader()`
  - Returns the angle from the iRobot Create chassis in milliradians (precise to 1 degree).
- Single Motor
  - `new RangeTrackSingleMotorPositionReader(int port)`
  - Returns the position of a DC motor in ticks, as measured by the CBC's back-emf circuitry.
- Single Servo
  - `new RangeTrackSingleServoPositionReader(int port)`
  - Returns the goal position of an RC servo in ticks.
- Time
  - `new RangeTrackTimePositionReader()`
  - Returns a timestamp in milliseconds.
  - An easy-to-use alternative to true odometry, which is still relatively accurate.
- Sum
  - `new RangeTrackSumPositionReader (vector<RangeTrackPositionReader> readers)`
  - Returns A + B, where A and B are the values of two other Odometry sensors .
  - Useful for calculating distance traveled by a differentially-steered robot chassis.
- Difference
  - `new RangeTrackDiffPositionReader (vector<RangeTrackPositionReader*> readers)`
  - Returns A - B, where A and B are the values of two other Odometry sensors .
  - Useful for calculating the angle of a differentially-steered robot chassis.
- Constant
  - `new RangeTrackConstantPositionReader(long value)`
  - Stores a constant value and returns it.
  - The value can be changed with a function call (yes, it's ugly):
    - `((RangeTrackConstantPositionReader*)(rt->GetXReader() ))->SetPosition(30); // X=30`
  - Useful for debugging or creating custom sensors.
  - In practice, you'll probably be using this one a lot.
- Null
  - `new RangeTrackNullPositionReader()`
  - Always returns 0.
  - Kind of pointless, but useful for debugging (it was also the first sensor driver written for RangeTrack).

- Range
  - ET
    - `new RangeTrackETRangeReader(int port)`

        - Returns the distance in millimeters from the Sharp GP2D12 ("ET") rangefinder.
        - Uses a lookup table, generated using a power regression, for superior speed and accuracy.
    - Sonar
        - `new RangeTrackSonarRangeReader(int port)`
        - Returns the distance in millimeters from the Maxbotix EZ-1 (CBC Sonar).
        - The EZ-1 returns a distance in inches; this is scaled to millimeters by the sensor driver (so the precision will never be better than 1 inch).
        - Still under development.
    - Raw Analog Range
        - `new RangeTrackRawAnalogRangeReader(int port)`
        - Returns the 10-bit analog-to-digital value of a sensor.
        - Useful for debugging.

The data logging is handled by its own C++ class, which logs four pieces of data each iteration: the X, Y, and Theta coordinates of the sensor (odometry sensors), and the range distance (a range sensor). By convention, all distances are in millimeters, and all angles are in milliradians (not following these guidelines may cause unintended results). The logger class contains a function called UpdateAll(), which logs one frame of data; the user program should call this function repeatedly when new data is desired. (The UpdateLog() function is available when the user wishes for only data logging to occur with no further calculations, which conserves CPU time.)

The ET driver uses two useful tricks to obtain good speed and accuracy. Because of the ET's use of triangulation to measure ranges, its analog voltage is not directly proportional to distance; it can be very accurately estimated with a power regression [8]:

$$d(s) = 2141.72055 s^{-1.078867}$$

Where *d* is range in centimeters and *s* is an 8-bit analog voltage.

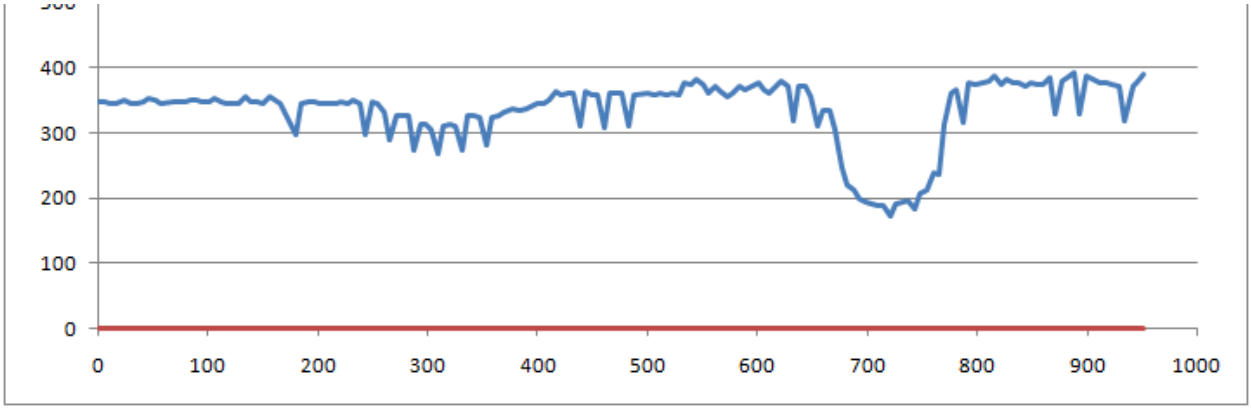
At the time when this code was being developed, the XBC was being used for Botball, and the Game Boy Advance was incapable of performing this floating-point math as fast as the data was coming in. We instead pre-calculated the range for every possible analog voltage using Excel, and generated a lookup table in a .h file which is much faster. For example, a single reading using the power regression took 17 milliseconds of calculation time on the XBC; the lookup table with identical accuracy took 12 *microseconds*. Some additional storage is necessary for the lookup table, but the XBC has 4MiB of flash, which is more than enough (less than 1MiB is used by the official XBC IC firmware).

The speed issue is probably less relevant on the 350MHz Chumby than on the 16MHz GBA, but we figured there wasn't much point in rewriting code that worked, and if it uses less CPU on the Chumby than it otherwise would, that's certainly not a bad thing.

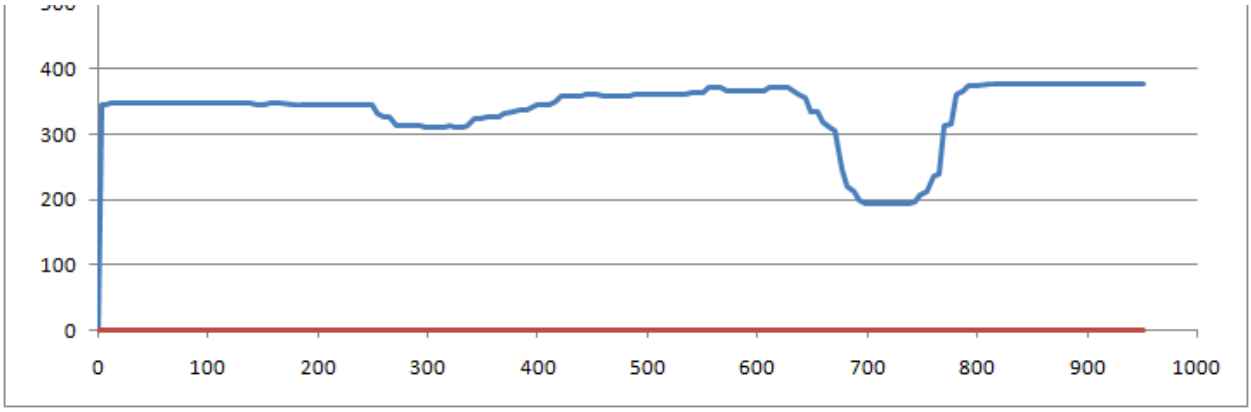
6 Noise Reduction

Rangefinders such as the ET are extremely noisy, so some post-processing is necessary to

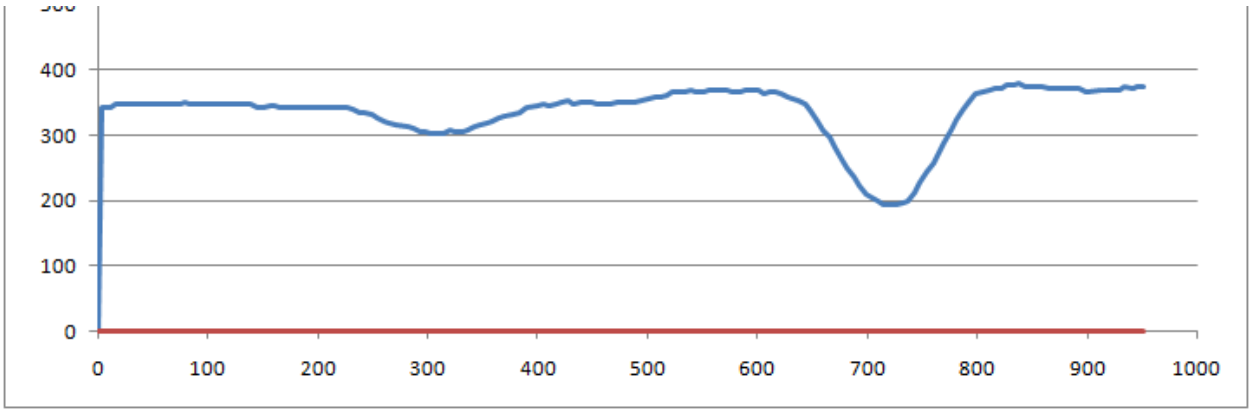
obtain usable data. Noise-reduction filters are implemented as drivers, similar to sensors. The two main types of filters are a median filter and a mean filter. Median filters are excellent at completely wiping out large noise, but do very little to low-level noise. Mean filters are the opposite: they almost completely eliminate low-level noise, but can be severely impacted by large noise. As a result of both this theoretical behavior and our non-exhaustive testing, a combination of a median filter and a mean filter, both with a diamater of 5 points, appears to yield the best results. A visualization of the various filter options is below (the graphs depict a wall, with a Pringles can very close to the wall at x=300mm, and another Pringles can farther from the wall at x=725mm):



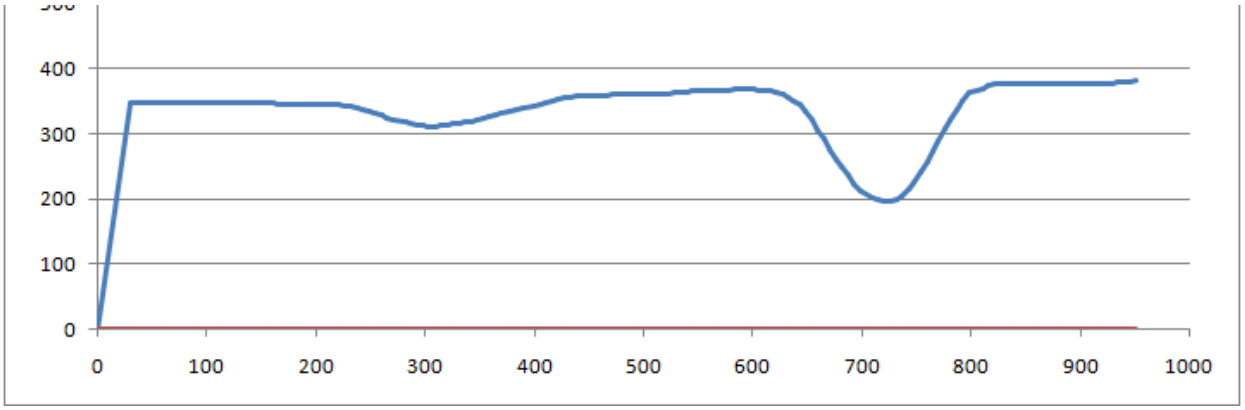
(No Filtering)



(Median Filtering)



(Mean Filtering)



(Median + Mean Filtering)

7 End of Part 1

That's it for Part 1. Part 2 will continue where we leave off here, covering what information can be obtained based on the filtered data visualized above. See you there!