Phase 3: Plotting and Ballistic Trajectory Prediction.

# Overview

In this phase, your team is responsible for the radar system’s data analytics. As implemented in Phase 2, the radar returns a series of range, azimuth, and elevation data points while an inbound missile resides within the radar’s sensor “wedge”. Based on this data series, your team must predict the future trajectory of the missile.

All key presses implemented in Phase 2 \*must\* still function properly. New key presses for this phase include:

1. Key “3”: Plot most recent data series acquired via radar
2. Key “4”: Using the most recent data series, fit a quadratic equation to this series. Sample this equation at equally spaced horizontal intervals for the entire predicted flight time of the observed missile. Assume the missile is always launched at z = 0 and impacts when z = 0. Plot this predicted path overlaid with the observed data.

All programmers \*MUST\* do a pull in the following repos to obtain updated source code:

aburn/engine – Then run cmake and build the INSTALL MinSizeRel and INSTALL Debug  
aburn/usr – This will place a folder called gnuplot inside of aburn/usr/shared/. gnuplot is required for this phase.

# Deliverables

## GNU Plot – Observed Trajectory Plotting

The bidding war between graph/plot vendors is over. The plotting provider is GNU Plot for this system. The vendor’s interface, “gnuplot-iostream”, encapsulates the power of GNU Plot within a C++ iostream. This eases interoperability between C++ data and its corresponding graph. gnuplot-iostream is a header-only library that is included in the Phase3 folder of the csce593 repo. The vendor’s support website is located at: <http://stahlke.org/dan/gnuplot-iostream/> . **Note:** I made a minor update to the vendor’s source code, be \***sure**\* to use my version, or you are likely to have issues making a plot appear, see SLN NOTES.txt for details.

Each team must learn how to use this API. Each team must apply sound software engineering principles to encapsulate the functionality of these plots into an interface conducive for this application (see Grading section). After a missile flies through the wedge (and before the 2 key is pressed to reset said missile), pressing the 3 key will plot the observed trajectory as shown below. The X-axis denotes horizontal range, in meters, to the radar sensor. The X-axis \*decreases\* from left to right. The Y-axis denotes the elevation, above ground in meters, of the missile. This graph should be regenerated upon pressing “3” (if the 2 key, which resets the missile state and removes its trajectory information, has not yet been pressed).

Your team will likely need to revise the lambdas you created in the GLView’s loadMap. For example, the onFilterFunc will likely need to store all observed data points in a more permanent location. This way, the “3” key and “4” key can iterate over the observations and transform those to data that can be passed to a plot method or to a plot with a prediction method. This same data store can also be utilized for deliverable 5 (discussed below).

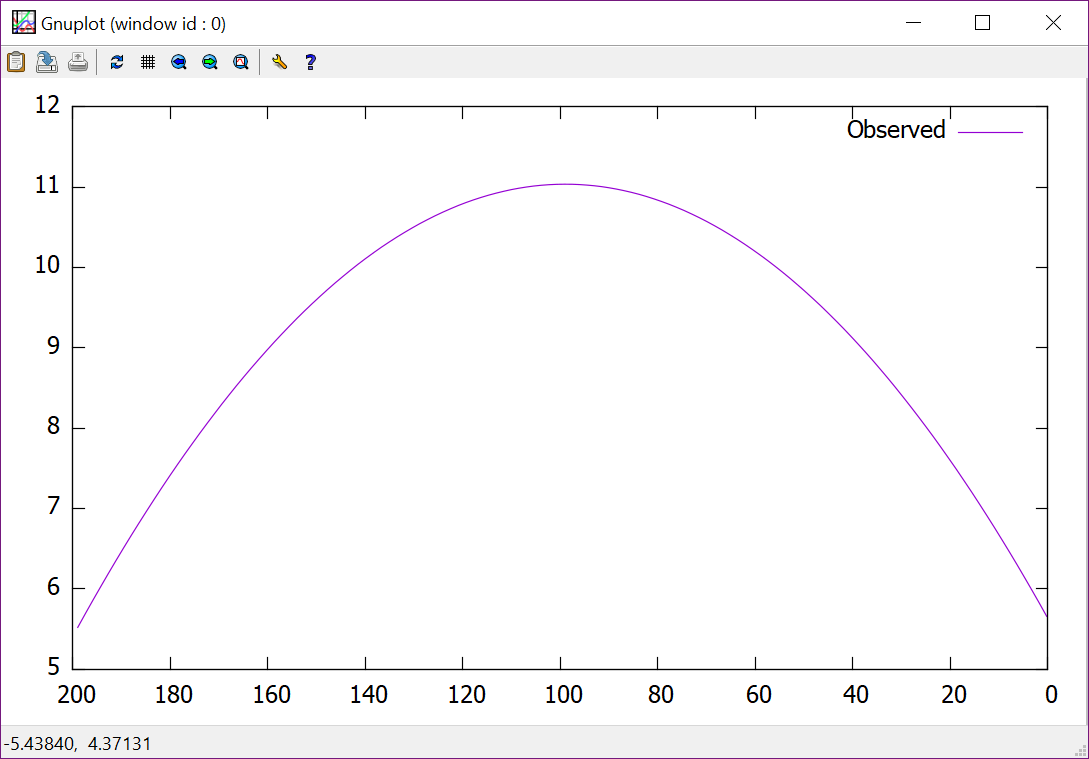


Figure 1. A Plot showing an observed path. Notice we only see ranging data from [0,200] meters. This is because our radar can only observe within those distances. Ensure these values are consistent with your conf file.

## GNU Plot – Predicted Trajectory of Entire Flight Path

While plotting is useful, we must develop a mathematical model to predict the inbound missile’s quadratic trajectory (ballistic trajectory). To this end, the repo also includes Vili Petek’s PolyFit.h, a helpful set of functions for curve fitting. As software experts with a solid mathematical background, you must figure out how to use PolyFit to generate a 2nd order polynomial that approximates the observed trajectory. PolyFit’s website is: <http://vilipetek.com/2013/10/07/polynomial-fitting-in-c-using-boost/> . Refresh yourself with things like, parabolas, second order curves, and the quadratic equation. Fully read the aforementioned link, if the link is down, a copy is already in the repo for your learning convenience (./Phase3Plotting/PolyFit/\*.pdf).

Once you have a mathematical function, we will plot said function. Ultimately, this will provide a target coordinate for our defender missile. However, our quadratic function is valid from neg infinity to pos infinity. Fortunately, the only portion we care about is the segment where the inbound missile is above ground (z >= 0). Therefore, you must find these two endpoints and plot the data within this domain, inclusively. HINT: The points at which a second order parabola cross 0 (the plane z=0) are the roots of the function (please fondly recall the quadratic equation from 8th grade and reminisce). Since you are deriving a continuous function from a discrete set of observed points, you will plot your predicted flight path as evenly spaced data points along the x-axis (see Figure 2). To do this, use linear interpolation, try searching the AftrBurner engine for “lerp”, and research this yourself. Figure 2, below, corresponds to the prediction of the plot shown in Figure 1. The predicted values are overlaid on the observed values.

When trying to create your first GNU Plot, the gnuplot-iostreams object takes a string as a constructor parameter. This parameter expects a relative path to gnuplot.exe. The engine already has a copy of gnuplot.exe located in "…/aBurn/usr/shared/gnuplot/bin/gnuplot.exe"… A convenient way to access this string is via the singleton “ManagerEnvironmentConfiguration::getGNUPlotBin()”.

Use my provided solution for comparison. You will have to integrate both gnuplot-iostreams and PolyFit.

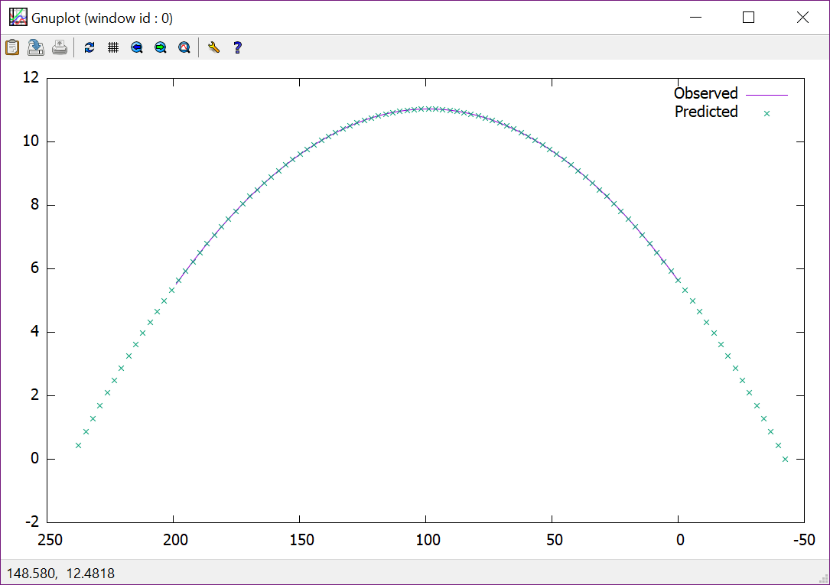
\*USE gnuplot-iostream and PolyFit FROM THE REPO – Do \*NOT\* re-download it. No mercy will be shown for questions related to not following this rule.

Figure 2. Notice how the predicted values begin with launch and end with the impact. The predicted values overlay the observed values. Notice the prediction tightly fits the observed data.

## Precise Control of Sensor via Conf File

Make the RadarDish the primary parent of the radar system. It’s position must be dynamically settable via the “radarDishPosition=(10,10,10)” conf file entry. This should be reflected when Shift-R is pressed. The sensor object remains a child of the RadarDish. Additionally, make the tower a child of the dish, the tower, however, must not spin when the RadarDish spins. One technique is to use WOGJoints between the two parts and counter rotate the tower by the opposite rotation as the dish – this way the tower remains stationary while the dish spins (thereby moving the sensor). The tower may float “above” the ground or embed itself deep within the ground if the conf setting implies this. See the figures below as reference. The important part is that dish->getPosition equals the radarDishPosition. This implies that sensor->getposition also equals radarDishPosition.



Figure 3. radarDishPosition=(10,10,0) Figure 4. radarDishPosition=(10,10,30)

## Integrating “Smart” Ribbons to Visualize Inbound Missiles

Your team must incorporate the ability to visualize the flight path of inbound missiles in real time. When any missile is in a FLYING state, it must employ a WOTrackingRibbonBase world object. This WOTrackingRibbonBase is composed of an MGLTrackingRibbon Model object to perform the actual drawing. At each position, an instance of an Aftr::TrackInfo (defined in MGLTrackingRibbon.h) can be added to the ribbon via addTrackPoint() to grow the ribbon. The ribbon should be added as a child of the flying Missile and each missile shall have its own ribbon. The ribbon’s history should be reset whenever the missile’s reset(); is invoked. Upon launch a “clean” ribbon should begin following the missile.

Our intel team has also captured an enemy missile and learned that they have embedded detectors that sense our radar’s radiation. For intel purposes, the general has ordered that our system clearly articulate the portion of each missile’s flight path that directly exposes it to the radar’s radiation.

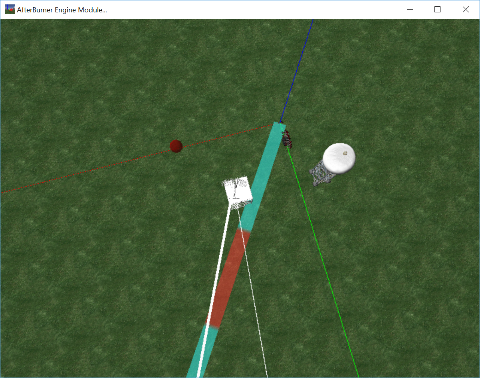
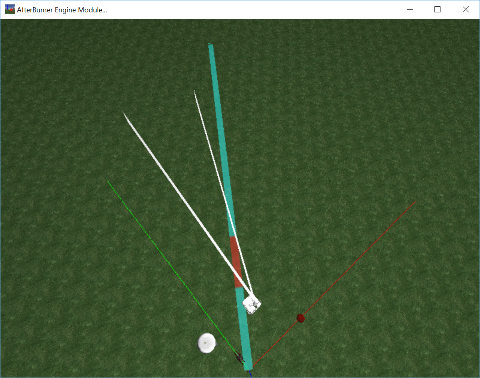


Figure 5. Visualization of Inbound Missile. Notice the portion of the flight path detected by the radar is colored red.

In response to this new intel, our Radar vendor has issued an update to their sensor API forcing a “minor” breaking change. The new API modifies the input to the sensor’s addPossibleTargetToRadarScanList() method. The new prototype is now:

void WORadarSensor::addPossibleTargetToRadarScanList( WO\* objToSearchFor, RadarCallback\_onObjectIlluminatedByRadar onRadarIlluminationOccurredCallback )

Within WORadarSensor.h, we find new defined alias:

using RadarCallback\_onObjectIlluminatedByRadar = std::*function*< void(WO\*) >;

This implies that the WORadarSensor::possibleTargets has now changed from an std::vector< WO\* > to a std::vector< std::tuple< WO\*, RadarCallback\_onObjectIlluminatedByRadar > >. Each tracked object pairs with a callback function that is invoked when a radar beams strikes the tracked object. This paired function is dispatched within the WORadarSensor’s onUpdateWO() method. The callback is designed to mutate the tracked missiles state such that it “remembers” it was recently struck by a radar pulse. When the missile subsequently appends its next Tracking Point into its ribbon, this state can be checked and cleared. In this way minimal coupling exists between the Radar and each entity that is tracked and can sense the Radar’s radiation when pointed at that entity..

Based on this new intel, the WOInterceptorMissile class must be extended to support 2 new public methods, one state variable, and the ribbon:

public:

/// Phase 3 A missile knows when radar is illuminating it

bool currentlyDetectsRadarBeams() const { return this->detectedRadarPulse; }

/// Phase 3 A missile knows when radar is illuminating it

void currentlyDetectsRadarBeams( bool detectedRadarPulse ) { this->detectedRadarPulse = detectedRadarPulse; }

private:

bool detectedRadarPulse = false;

/// Plots 3D motion of missile in FLYING state (OGL 2.1 compat ribbon)

WOTrackingRibbonBase\* ribbon = nullptr;

Using the information above, your team can make the Ribbons change color accordingly.

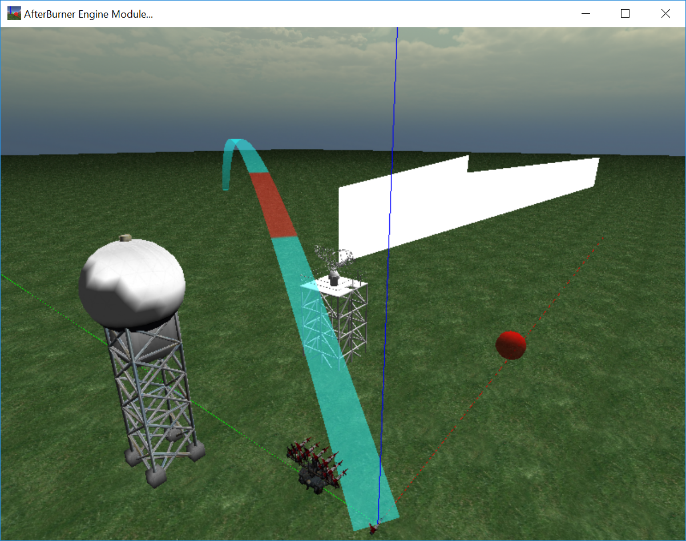


Figure 6. Yet another beautifully tracked inbound missile and its visualized exposure to radar radiation.

# GNUPlot - Asynchronous Automatic Plotting

For deliverables 1 & 2, although the sensor was tracking inbound objects in real time, a user had to press the “3” or “4” key to draw the latest version of the plot. This works reasonably well. However, for deliverable 5, we want our system to *automatically* draw an updated plot each time new sensed trajectory information is available. This feature can be enabled or disabled by pressing the “5” key (see my solution as an example).

This asynchronous plot will be completely independent of deliverables 1 and 2 – BOTH in terms of source code and grade. It makes sense to write this code \*after\* you have completed parts 1 and 2. This deliverable will employ an asynchronous Active Object Pattern that encapsulates its own thread and its own instance of a Gnuplot (see ActivePlotter.h). The AOP thread will be responsible for performing/executing the expensive GNUPlot instructions. The AOP’s thread will simply spin in a loop pulling the latest data from a work queue. Each iteration of the AOP thread will execute that work. Your team must update the naïve implementation given in the constructor of the Active object defined in ActivePlotter.h.

To create the AOP object, consider placing code similar to the following in your GLViewDefenseDaemon.h file:

//InParam is an alias that defines how the user chooses to package the plot data that GNUPlot

//will consume. This makes the most sense to set it equal to the same data structure you pass

//in

using InParam = std::vector< ... >; //What type should go in the ...

ActivePlotter<InParam>::Worker\_Func\_Prototype asyncPlotFunc;

std::unique\_ptr< ActivePlotter<InParam> > realTimePlotter = nullptr;

Here, InParam is an alias that defines the template type defining half of the input data we must pass to our ActivePlotter’s worker lambda. This is the lambda that actually calls GNU Plot. The variable asyncPlotFunc is the lambda that is actually executed within the AOP’s worker thread. This is the lambda method that will take as input an InParam as well as a Gnuplot and will draw a plot of the data inside InParam. Within the GLViewDefenseDaemon.cpp, I would recommend defining the body of asyncPlotFunc at the bottom of loadMap(). Lastly, when the 5 key is pressed, this will enable or disable asynchronous, real time plotting via GNUPlot. For reference, my implementation of the SDLK\_5 is given below:

else if( key.keysym.sym == SDLK\_5 )

{

static bool showAsyncPlot = false;

showAsyncPlot = !showAsyncPlot;

if( showAsyncPlot )

{

this->realTimePlotter = std::make\_unique< ActivePlotter<InParam> >( this->asyncPlotFunc );

std::cout << "BEGINNING ASYNC PLOTTING OF ALL OBSERVATIONS...\n";

}

else

{

this->realTimePlotter = nullptr;

std::cout << "HALTING ASYNC PLOTTING OF ALL OBSERVATIONS...\n";

}  
 }

# Grading

Cleanly integrate your new code with the rest of your solution. Ensure you adhere to guidance given in software quality attributes in terms of maintainability, testability, usability, low coupling, high cohesion, sound class decomposition, etc. Your team will be graded on these metrics as well as the correctness of your solution. As your team progresses through this phase, important design decisions, sticking points, each individual’s time invested, morale, and other observations are to be summarized in a presentation and delivered on the due date. This presentation also serves as a means to justify your design decisions. If a specific decision initially appears counter intuitive, justify its rationale in your talk to explain your reasoning. Be thoughtful and inclusive with these items as it provides me with insight to your development processes and abilities.

See “Phase 3 Report.pptx” for a more descriptive rubric.