# ECE 275 Assignment 3

**DUE DATE:** Tuesday, November 4, 11:59PM, ~~Tuesday, October 28, 11:59PM~~

In this assignment, you will create an application that analyzes the behavior of an autonomous vehicle from the output of its GPS/IMU recorded along its journey. The application will read in NMEA sentences from log files output by the vehicle and record GPS information (latitude, longitude, and accuracy) and the time (in weeks and seconds) based on one specific NMEA sentence type. It should store this information within a class that represents the state of the vehicle at that second. Finally, the program will analyze the data to determine the operating state of the vehicle’s GPS/IMU at a range of specific locations the vehicle visited on its journey. The program will output an alphabetized list of these destinations that it visited based on GPS coordinates, and the state(s) of the GPS/IMU at that location (in order).

**Note:** This program is motivated by research done with the CAT Vehicle (http://catvehicle.arizona.edu/) in the ECE department.

The assignment name for this assignment is: **gpsimu**

## Command-line Arguments

Your program must be capable of utilizing a command-line argument to specify the log file and the output file.

gpsimu logFile outputFile

Your program must ensure the user has correctly provided the required command-line arguments and display a usage statement if the provided arguments are incorrect.

## Input Text Files

For this assignment, the input text file will consist of an extension of NMEA sentences that is proprietary to NovAtel GPS units with tightly coupled inertial sensors (for brevity we will call these NMEA sentences). You will need to parse these strings appropriately. The NMEA sentences begin with a hash symbol (#) and consist of a header separated by data with a semicolon; individual values within the header and the data sections of a sentence are separated by commas. This program will need to parse only #INSPVAA sentences. However, there are other possible NMEA sentences that you will have to ignore. Examples of #INSPVAA are as follows:

#INSPVAA,COM1,0,54.5,FINESTEERING,1748,502500.800,00000000,5615,4674;1748,502500.794775700,0.000000000,0.000000000,0.000000000,0.000000000,0.000000000,0.000000000,0.000000000,0.000000000,0.000000000,INS\_ALIGNING\*3fe523b6

Tables 1, 2, and 3 at the end of this document show the breakdown of the NMEA sentence for #INSPVAA. Note that these are basic descriptions and are sufficient to complete the assignment. If you are curious, there are other sources of information on the web that you are encouraged to look up, including lots of other proprietary GPS sentences that could be included in these files. Note that the fields that are important for this assignment have been *emphasized in that table*.

Please note that for this assignment you ***DO NOT*** need to check that input file matches this format. Instead, you may simply ***assume*** that it will. In other words, your program can assume that if any data are in the file read in with the indicator for that sentence, then those data will conform to that sentence structure, and that lines for other sentences can be ignored.

**Parsing**

You should have a class called NMEAParser that handles all of the parsing for this assignment with static functions. You should have a header and source file for this class. The class definition should appear as follows (you should be able to determine the functionality based on the function names, parameters, and return type, but there may be ambiguities so please ask questions on Piazza); make sure you add your own comments for each feature of the class:

class NMEAParser {

public:

// If unable to read file, the vector will be of size 0

static vector<GPSIMUState\*> parseLogFile(string filename);

static GPSIMUState\* parseNMEASentence(string sentence);

private:

// These are private because there is no need for them in a ‘static’

// class.

NMEAParser(void);

~NMEAParser(void);

};

**Hint**: Try calling the parser once you’ve created a VehicleJourney object, and set its states member when parsing the log file. You’ll then be calling operations of VehicleJourney to complete the remainder of the assignment and to generate new vectors of objects.

**GPS/IMU State**

The state of the autonomous vehicle’s GPS/IMU can take on one of three values. Init, Ready, and Standby. The unit ~~will always start in the Init state and it[[1]](#footnote-1)~~ transitions between states based on the value of the INS Status (field 13 of the #INSPVAA sentence). Upon receipt of a #INSPVAA sentence the state of the GPS/IMU should be set to the value based on the information in Table 4 shown at the end of this document.

## Location Detection

Your program will have to examine the GPS data from the NMEA log files to determine where the vehicle has been on its journey. In order for the vehicle to be considered “at” a location, it must be within ±0.002 degrees in both latitude and longitude. Table 5 shows a list of GPS locations around Tucson that the vehicle might visit.

## Data Structures

In this assignment you will make several C++ classes for data storage. The first of these will store information about an individual GPS/IMU state. You should have a header and a source file for each of these classes. It should be defined as follows:

class GPSIMUState {

public:

// Constructor and Destructor

GPSIMUState(void);

GPSIMUState(std::string sentence);

~GPSIMUState(void);

/\* Implement other methods as you see fit \*/

/\* Mutators for variables are below \*/

// mutators for week

void setWeek( unsigned long week ) { this->week = week; }

unsigned long getWeek( ) const { return this->week; }

// mutators for seconds

void setSeconds( double seconds ) { this->seconds = seconds; }

double getSeconds( ) const { return this->seconds; }

// mutators for latitude

void setLatitude( double latitude ) { this->latitude = latitude; }

double getLatitude( ) const { return this->latitude; }

// mutators for longitude

void setLongitude( double longitude ) { this->longitude = longitude; }

double getLongitude( ) const { return this->longitude; }

// mutators for state

void setState( std::string state ) { this->state = state; }

std::string getState( ) const { return this->state; }

// mutators for location

void setLocation( std::string location ) { this->location = location; }

std::string getLocation( ) const { return this->location; }

private:

unsigned long week; // The week number from the NMEA sentence.

double seconds; // The seconds from the NMEA sentence.

double latitude; // The latitude from the NMEA sentence.

double longitude; // The longitude from the NMEA sentence.

std::string state; // The GPS/IMU State at this point.

std::string location; // The location name for this position.

};

Next is a class that will store information about locations that the vehicle visits along its journey.

class Visit {

public:

// Constructor and Destructor

Visit(void);

Visit( std::string location, double startTime, double endTime );

~Visit(void);

// Compare two visits together to see which is before the other.

// Returns true if v1 is before v2, false otherwise.

// A visit is before another if its location is alphabetically before

// the other visit’s location.

// A visit with the same location name as another is before that

// location if its startTime is before the other visit’s startTime.

static bool compare(Visit\* v1, Visit\* v2);

/\* Implement other methods as you see fit \*/

/\* Mutators for variables are below \*/

void setStartTime( double startTime ) { this->startTime = startTime; }

double getStartTime( ) const { return this->startTime; }

void setEndTime( double endTime ) { this->endTime = endTime; }

double getEndTime( ) const { return this->endTime; }

void addState( std::string stateString );

std::vector<std::string> getStates( ) const { return this->state; }

void setLocation( std::string location ) { this->location = location; }

std::string getLocation( ) const { return this->location; }

// returns the string for this visit

std::string getVisitString( ) const;

private:

// Note that you can assume the week will always

// be the same for start and end times.

std::string location; // The location name for this visit.

double startTime; // The start seconds for the visit.

double endTime; // The end seconds for the visit.

std::vector<std::string> state; // The ordered list of states at this location.

};

Finally, here is the definition for the journey class that stores all of the information about the vehicle’s journey.

class VehicleJourney {

public:

// Constructor and Destructor

/\* also constructs the interesting locations vector to consider in this journey \*/

VehicleJourney(void);

~VehicleJourney(void);

/\* Implement other methods as you see fit \*/

/\* Analyze journey calculates the visits and sorts them \*/

void analyzeJourney( );

/\* writes the output file \*/

bool writeOutputFile( std::string filename );

/\* returns the output strings for printing out \*/

std::string getVisitsAsStrings( ) const;

Location\* findLocation(GPSIMUState\* state);

/\* Mutators for variables are below \*/

void setStates( std::vector<GPSIMUState\*> states ) { this->states = states; }

std::vector<GPSIMUState\*> getStates( ) const { return this->states; }

private:

/\* looks through the state vector and determines the visits, including

setting all values for each visit \*/

void determineVisits( );

/\* sorts the visits vector after it is populated \*/

void sortVisits( );

// member variables

std::vector<GPSIMUState\*> states; // The list of GPS locations and states.

std::vector<Visit\*> visits; // The locations that were visited.

// if you create this list when you construct your vehicle journey, then

// it may be easier to check whether you are in a location or not

std::vector<Location\*> locations;

};

Note that we are leaving the majority of the function implementations up to you. We will be looking for good practices as much as you know them (constructors, destructors, properly named functions, etc.), but the details are up to you. *You can also add more attributes and operations to each class, if you want to.*

**Pro tip**: You can easily sort a C++ vector using the std::sort() function and a function pointer to a comparator function that you’ve defined.

## Output File Format

This assignment should output a file with information on the locations that the vehicle has visited along its journey. It will output these locations in alphabetical order first, then by time. That is, if the vehicle visited a unique location more than once on its journey then it should output information for each time it visited that location and sorted with the earliest visit to that location first. The format for each output entry is as follows

<location>\n

<startTime> to <endTime>\n

States: <state1>, <state2>, … ,<staten>\n

\n

All white space above is either new lines (‘\n’) or spaces. <location> is the name of the location as indicated by Table 5 below (do not print out unknown locations). The <startTime> and <endTime> should with a precision of 8[[2]](#footnote-2). <statei> indicates the state as indicated by the second column of Table 4. Note that all states of each visited location should be displayed. If the GPS/IMU stays in the “Standby” state for an entire Visit, then “Standby” would be printed once (e.g., States: Standby\n)

|  |  |  |
| --- | --- | --- |
| **Table 1: NMEA Sentence Header Field Descriptions** | | |
| **Index** | **Field** | **Description** |
| *1* | *Sync* | *The hash symbol (#).* |
| *2* | *Message* | *The sentence identifier (e.g., BESTGPSPOSA or INSPVAA).* |
| 3 | Port | The data port that the sentence arrived on (e.g., COM1). |
| 4 | Sequence Number | Counts down from n-1 if the sentence is a part of a larger data transmission. |
| 5 | % Idle Time | The minimum percentage of time the processor is idle between successive messages with the same ID. |
| 6 | GPS Time Status | Indicates the quality of the GPS time. |
| *7* | *Week* | *The GPS week number.* |
| *8* | *Seconds* | *The number of seconds since the start of the GPS week.* |
| 9 | Receiver Status | An eight digit code representing the status of various components. |
| 10 | Reserved | Internal use only. |
| 11 | Receiver Version | The build number of the receiver software. |
| 12 | ; | Indicates the end of the header |

|  |  |  |
| --- | --- | --- |
| **Table 3: #INSPVAA Sentence Field Descriptions** | | |
| **Index** | **Field** | **Description** |
| 1 | Header | The header described above. Ends with a semicolon. |
| *2* | *Week* | *GPS Week.* |
| *3* | *Seconds* | *Seconds from the start of the GPS Week.* |
| *4* | *Latitude* | *The latitude component of the GPS data.* |
| *5* | *Longitude* | *The longitude component of the GPS data.* |
| 6 | Elevation | The elevation component of the GPS data. (Mean high above sea level). |
| 7 | North Velocity | Velocity in a northerly direction (a negative value implies a southerly direction) |
| 8 | East Velocity | Velocity in an easterly direction (a negative value implies a westerly direction). |
| 9 | Up Velocity | Velocity in an up direction. |
| 10 | Roll | Right handed rotation from local level around Y-axis in degrees. |
| 11 | Pitch | Right handed rotation from local level around X-axis in degrees. |
| 12 | Azimuth | Left handed rotation around Z-axis Degrees clockwise from North. |
| *13* | *Status* | *INS Status* |
| 23 | xxxx | 32-bit CRC (ASCII and Binary only) |
| 24 | [CR][LF] | Sentence Terminator |

|  |  |
| --- | --- |
| **Table 4: GPS/IMU State Changes** | |
| **ASCII INS Status** | **GPS/IMU State** |
| INS\_INACTIVE | Init |
| INS\_ALIGNING | Init |
| INS\_SOLUTION\_NOT\_GOOD | Standby |
| INS\_SOLUTION\_GOOD | Ready |
| INS\_BAD\_GPS\_AGREEMENT | Standby |
| INS\_ALIGNMENT\_COMPLETE | Ready |

|  |  |  |
| --- | --- | --- |
| **Table 5: GPS Locations of Note** | | |
| **Location Name** | **Latitude** | **Longitude** |
| Fry’s | 32.251223 | -110.961524 |
| ECE Back Lot | 32.235599 | -110.953088 |
| University Boulevard | 32.231672 | -110.959131 |
| 4th Avenue | 32.227751 | -110.965590 |
| Eegee’s | 32.235891 | -110.934919 |
| CAT Vehicle Testing Lot | 32.217875 | -110.914389 |
| In-N-Out | 32.221842 | -110.957492 |
| Elliot Electronics | 32.205207 | -110.959206 |
| Unknown | Any other Latitude | Any other Longitude |

1. You cannot depend on the unit always starting in the Init state; clarifying that here. [↑](#footnote-ref-1)
2. Note: this is a change from a previous version of the assignment, which required 8 characters. [↑](#footnote-ref-2)