Software Development Policy/Procedures Manual for Autonomous Vehicle (AV)

Joshua Buhain

First Version, written January 25, 2023

Table of Contents

1. Introduction

In this software procedures manual, we will address important ideas that we as a team will apply during the development of our autonomous vehicle software. This manual is made up of three main sections: coding/naming conventions, development process, and data management process. Our coding and naming conventions will aim to provide readability and good clarity to our code and file names. Our development process will go over our plan to implement Test-Driven Development approach and how that will translate into how we write and test our code and software. Our data management process will go over our plan to implement Continuous Integration and how we will store and back up our data.

1.1 Design Space Exploration

For our sensors, we are planning to use cameras, lidars, and radars. Each type of sensor has its own drawbacks and each of these sensors complements the other's weaknesses. For example, Lidar can be sensitive to the environment, like temperature and fog levels[1]. Used in weather systems or even in space, radar sensors can work very well under extreme conditions. However, this comes with the cost of weaker movement detection and a shorter range. Camera sensors are good at detecting letters, traffic signs, and most importantly, people and animals[2].

For the route planning algorithm, we are planning to build off of the A-star search algorithm. For the world perception and decision-making algorithm, we will combine the data taken from the sensors mentioned above and throw them into a deep learning network like CNN (Convolutional Neural Network).

For our code base, we are planning to use Python for writing our scripts for data processing and building off of that, write code in a Notebook like Jupyter to train our model. We will use C++ for doing a lot of the heavy work of our algorithms. We are planning to use Matlab to test our autonomous vehicle software.

[1] Source from Fierce Electronics website
[2] Source from Lidar Radar website

2. File Organization

2.1. File Extensions formats

File extension formats are self-explanatory except for languages such as C++ which has multiple names for file extensions. C++ source files should have a '.cpp' extension and C++ header files should have a '.h' extension.

2.2. Directories Naming convention

Source directories will be named 'source.'
Header directories will be named 'include.'
Object directories will be named 'object.'
Binary directories will be named 'bin.'

2.3. CSV File and Simulation files Naming Conventions

Thousands, if not tens of thousands, of CSV files, will be created during this project. Descriptive CSV files, although intimidating to look at, can be easier to retrieve and identify. Having a descriptive filename tag will save lots of money and time in the long run.

CSV naming form
ACTVT_CONFIG_SENSORID_YYYY-MM-DD-HH-mmTMZ_YYYY-MM-DD-HH-mmTMZ.csv

Example of Decoding filename
HACCNV_A00001_LID12345_2023-01–22_09-30PST_2023-01-22_10-00PST.csv

Example Filename Details

- Activity Type: HACCNV (Highway-Adaptive-Cruise-Control-No-Vehicle Ahead)
- Configuration: A00001 (Configuration-A Trial 0001)
- Sensor ID: LID12345 (Lidar#12345)
- Start Time: 2023-01–22, 09:30 PST
- End Time: 2023-01-22, 10:00 PST

Important Notes

- End time is optional for some files.
- The number of characters for each section in the filename can vary depending on the situation. The number of characters reserved for start/end times is fixed.
- All letters are capitalized.
- Abbreviating words should be avoided.

2.4. File Naming Conventions (Non-CSV)

We will have a simple naming convention for non-CSV files. Please do not start files with a number. Use underscores between strings, and only use one underscore at a time. The first letter of each string will always be capitalized. To maximize clarity, refrain from using abbreviations unless the word is very long (>15 characters).

3. Indention, Whitespace, and Newlines

We will be prioritizing readability over compactness/minimalist code. Spaces have a negligible effect on the performance of code and can improve readability which will save us time and money in the long run.

3.1.1. Indention: Tab stops

We will be using standard 4 spaces per level. We believe that 2 spaces make can make code look too condensed and we are not using languages like Javascript which have many more indents on average than C++ or Python. On the other hand, we believe that having tab stops at 5, 6, or 8 causes too big of a spacing.

3.1.2. Indention: Use Tabs or Spaces?

We will be using tabs for our indentation method instead of spaces. Spaces are arguably better but this will take more work to enforce. Most of our programmers prefer tabs for their convenience, and old habits die hard. We cannot risk having intermingled tabs/spaces as this can lead to critical bugs for programs written in languages that are sensitive to indents like Python.

3.1.3. Indention: Indentation Style

We will be adopting K&R (Kernighan & Ritchie Style) as our code indentation style. Out of all the standard indentation styles, K&R is the style that is most familiar to our programmers. Therefore, this style will probably be the fastest to read and scan.

3.2. Whitespace

We will be adding whitespace between identifiers and parentheses, and between identifiers and operators. This will be useful when debugging operation-heavy programs in Matlab. We believe that this will result in lower visual clutter which increases readability.

3.3. Newlines

We will be adding a new line at the end of our source codes. This is purely for convention and has no performance drawback.
We will only use ‘\n’ as the new line sign only.

3.4. Character Limit Per Line

We will be restricting all lines to 80 characters to improve readability and presentability.

3.5. Automatic Formatting Tools

We will be using AStyle to help format our code. Make sure that it is configured properly to our programming style. Unfortunately, AStyle is not available in some of the languages that we are planning to use (Matlab, for instance) so do not rely too much on this tool.

4. Coding Comments

We will be aiming to write comments that provide clear and meaningful explanations for our code. Comments can add visual clutter to the program but they are necessary when maintaining the code base. We will try to make the formatting of our comments as least visually disruptive as possible.

4.1. Multiline Comments

We will be using the single-star aligned format for multiline comments.

4.2. Single-Line Comments

We will be only writing single-line comments before the line of code we are commenting on.

4.3. Commenting Tools

We will be using Doxygen as our commenting tool as it is compatible with all the languages that we will potentially be working with for this software. Do not use any other commenting tools.

5. Programming Naming Conventions

Many programmers will be collaborating on our codebase so we have decided to enforce uniform naming conventions across all languages we will be using. Names should be meaningful and describe their use. For example, function names should be meaningful and describe their use like getName() or calculateError(). To maximize clarity, avoid using abbreviations when naming. Only use abbreviations for very long words.

	Naming Convention
Functions	camelCase
Type Names	t_camelCase
Defines	SCREAMING_SNAKE_CASE
Struct Names	s_camelCase
Class Names	c_camelCase
Data Members	d_camelcase
Enum Names	e_camelCase
Enumerated Values	enum classes with _camelCase
Local Variables	h_camelCase
Global Variables	g_camelCase
Parameters	lowercase

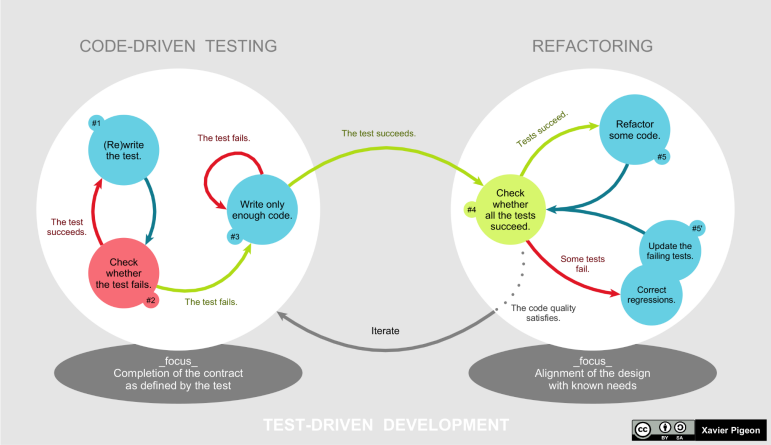
6. Development Process

Writing AV software demands a high level of coding integrity as the stakes are very high. We believe that adopting Test-Driven Development Methodology will help maintain the quality of our code to the highest quality while maintaining a fast development time. We will be describing what this methodology is and what that means in our development process.

6.1. Test-Driven Development Methodology

We will be taking a Test-Driven Development (TDD) approach when it comes to our process for developing our software. The TDD approach is about writing code as a response to test cases. This approach will lead to more robust and correct code since

we are only coding for what is necessary to complete tests. And since our code is a response to test cases, they will be easy to test.



TDD Chart from Wikipedia.[3]

6.2. Unit/Integration Testing with TDD Methodology

We will make sure that our unit tests are well-defined. And when writing to complete a unit test in TDD, it is important that the current code is a solution to only one particular test case, and not for multiple. Failing to do so is bad practice and will make our programs less robust.

When creating a new test, we must make sure that it *fails*. Otherwise, we will have to redo or revise this test. Then, we provide a solution to it. This solution should aim to be simple and only solve that particular test. After the solution is completed, we must run all possible tests and make sure that all tests made up to this point are passing. If not, then this solution is affecting other cases and must be revised. After passing the integration tests, the code is then refactored if necessary, and integrated into the code base. Rinse and repeat.

6.3. TDD Methodology for Building Algorithms

We will be developing a lot of algorithms during the initial stage of this project. To develop algorithms using TDD, we first create new tests. For instance, say we are refining our Dijkstra algorithm for route planning. We would create tiny new features like calculating the shortest distance in different units. These tests correspond to new small functions/requirements for our algorithms. We make sure that these tests fail when testing this algorithm in our simulation, which will probably be done in MatLab. Then, we write code that achieves this. Once the unit test passes, we then create integration tests to make sure that our newly added code is not affecting other functionalities.

[3] Test-driven development. (2022, December 27). In Wikipedia. https://en.wikipedia.org/wiki/Test-driven_development

6.4. Testing using Simulations

One of the main ways we will be testing our software is through MatLab simulations. Although MatLab is not fully confirmed yet in our code base, it will probably be likely. It offers ready-made tools for AV simulations.

For our software to pass a driving simulation, it must choose the safest action for all configurations of a simulation case. For example, say our scenario is “maintaining a proper speed next to a vehicle on the highway” we will have different configurations such as where our vehicle is next to a truck, a motorcycle, or a van, or different environment configurations such as rain or fog. When doing integration testing, we will throw the updated program into a wide variety of test simulations.

Once our software correctly handles the simulation, we will then do field testing with our vehicle. Testing with the car in real life will require that the software is meticulously tested through simulations.

6.5. Autonomous Driving Scenarios

There are a lot of driving scenarios that we will be testing, which will be listed out in a separate document in which our experts will mine data from DMV websites or other educational driving websites and classify these scenarios. We will separate these scenarios in terms of “difficulty” and order them in levels from 1 to 10, with 1 being described as a basic scenario and 10 being a more advanced driving scenario. We will work our way up in developing these scenarios to be handled by our software.

6.6. Algorithm Testing

We will be establishing certain thresholds for our algorithm tests. For instance, say we are trying to measure the performance of our perception algorithm’s ability to draw bounding boxes around people. We will be comparing the resulting bounding box with the expected bounding box area for each object. If the comparison is 95% accurate and our bounding box threshold is 98%, then this is an example of a failed test. With the technology of today, getting 100% accuracy on our perception algorithms is very unlikely but we will try to get this accuracy as high as possible.

7. Data Management

7.1. Continuous Integration

Will be adopting the Continuous Integration method for maintaining our codebase up to date amongst our team. We hope that this will speed up the production process by automating the build process after committing new changes to the codebase. This will

mean that everyone on the team will be committing their work to Git at least once a day cutting down the merging process time. We are planning on using either App Veyor or Jenkins CI as our continuous integration software as they both support GitHub.

7.2. GitHub Repositories

We will be using GitHub to store our codebase. We will divide the software into several parts with each part having its own repository. Some teams are going to be in charge of some parts of the software. This way, each project can scale dependently, and it's easier to track down the commits for each part. When one project relies on another, for example, a decision-making model relies on the perception algorithm, we will use git submodules to reference the perception algorithm in our decision-making model.

7.3. Storing Simulation/Sensor/Config Data

We will be storing our CSV data locally with a local server. Having a local server will mean fast accessing and uploading data to storage.

7.4. Backup System

We are planning to back up our data in the cloud using cloud web services. What we currently have in mind is using the AWS Backup application to store our sensor data and simulation data. AWS has great scalability and is well-known to be reliable. Reliability is very important because losing simulation files could mean wasted computing time and money. We are planning to back up our data at least once every 2 weeks into AWS.

8. Version and Change Log

The current version of this manual was created on January 25, 2023. This is the first released version of this procedures manual. This manual will be updated over time and the changes that were made will be documented in this section.