The purpose of my project aligns with the example presented in the paper, which is to detect vehicles on fixed open roads using monitoring systems. My focus, however, is on identifying the license plate numbers of vehicles from surveillance footage. I will record the total time that each vehicle with a corresponding license plate remains within the monitoring range. Knowing the total length of the monitored road, we can calculate the vehicle's speed and determine if it is exceeding the speed limit.

I employed three distinct detection methods for this project. The first method utilized Optical Character Recognition (OCR) without relying on an Automatic Identification Database (AIDB) to identify vehicles. The second method incorporated **AIDB** as a means of optimizing recognition speed, while maintaining identical parameters as the first method. The third method employed **Approximate Queries**, as discussed in the paper, which further improved recognition speed at the expense of some accuracy.

In this report, I will present the algorithms for each of the three methods, as well as the recognition results and recognition speed from multiple simulations. The surveillance footage used in this project was not obtained from actual recordings; instead, I employed word cards randomly appearing on screen to simulate vehicles sporadically driving on open roads.

```
pip install opencv-python-headless
```

For the first method, I employed the Vedio_to_image.py script to extract frames from a video at a specified frequency and save them in a designated folder named "\images". The frame extraction frequency was set to 15 frames per time.


```
pip install pytesseract Pillow
```

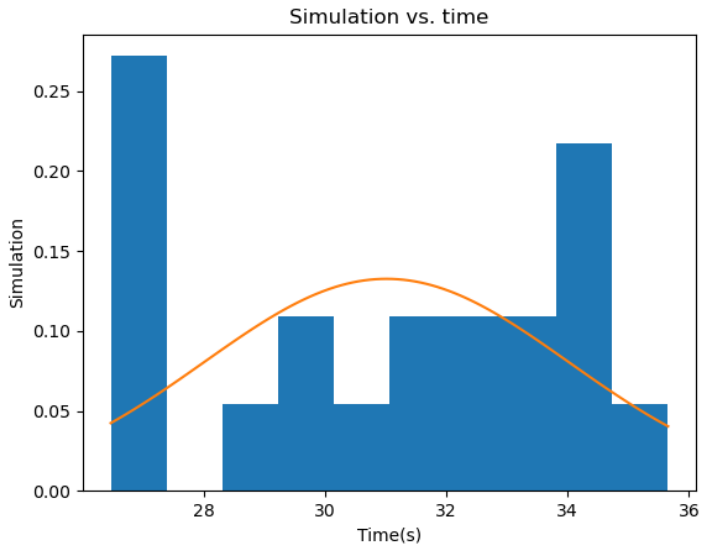
Next, I used Optical Character Recognition (OCR) in the Base_tables.py script to recognize text in images and output the data to an SQLite database. I utilized the Tesseract OCR library and the Python Imaging Library (PIL) for image processing, while the sqlite3 module was used to create and manage the database (referred to as **DataModel** in the paper). The first column served as the primary key, the second column contained the image names, and the third column held the "car number".

```
10|frame0135.png|
11|frame0150.png|
12|frame0165.png|
13|frame0180.png|
14|frame0195.png|
15|frame0210.png|verbiage


16|frame0225.png|verbiage


17|frame0240.png|verbiage
```

To group the same "car numbers" in the "basetable" table, I performed database processing tasks such as "group" in SQL using the User_defined_metadata.py script. The output's first column was the primary key, the second column consisted of the grouped text, and the third column displayed the count of distinct "car numbers" that appeared.

```
sqlite> select * from User_defined_metadata;
1|accessory|8
2|compensation|6
3|notation|3
4|verbiage|16
```

Finally, I replicated this process for 20 simulations and printed the distribution of the time it took for each run. I fitted the results using a normal distribution. The total time for processing four cars was 33 frames.

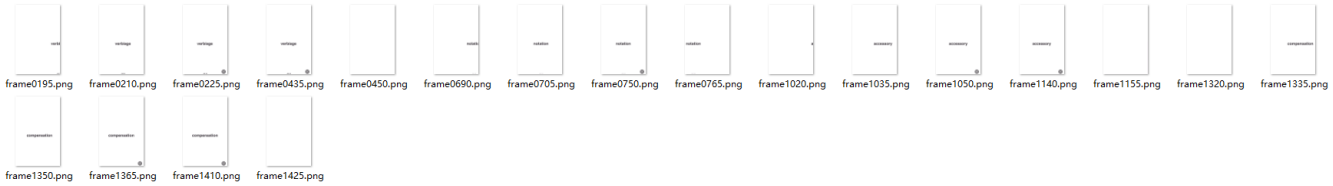


For the second method, the Vedio_to_image.py script remains the same. However, in Base_tables.py, I implemented a similarity test for every pair of adjacent screenshots. When a car passes through the monitored area, a difference occurs. After all, on most open roads, the time required to identify license plates is relatively short, and most of the monitoring is focused on empty roads. Similarity testing significantly reduces the number of images needing to be scanned during subsequent image recognition. Simultaneously, the recorded file name serves as a **timestamp**, indicating a strong connection between the image and the database. To ensure algorithm accuracy, I also recorded the previous frame that exhibited a change.

```
sqlite> select * from basetable
   ...> ;
1|frame0195.png
2|frame0210.png
3|frame0210.png
4|frame0225.png
5|frame0435.png
6|frame0450.png
7|frame0690.png
8|frame0705.png
9|frame0750.png
10|frame0765.png
11|frame1020.png
```

```
sqlite> .schema
CREATE TABLE basetable (id INTEGER PRIMARY KEY, image_name TEXT);
CREATE TABLE Mlmappings (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        image_name TEXT,
        text TEXT
    );
CREATE TABLE sqlite_sequence(name,seq);
CREATE TABLE User_defined_metadata (
        id INTEGER PRIMARY KEY,
        text_grouped TEXT,
        difference INTEGER
    );
```

Building upon the **base table**, we proceed to our **ML model mappings**. This time, I first copied the tested images, as per the second column in the "basetable", to the "\images_preprocessed" folder, and used the same OCR method from the first approach in mlmapping1.py and mlmapping2.py.

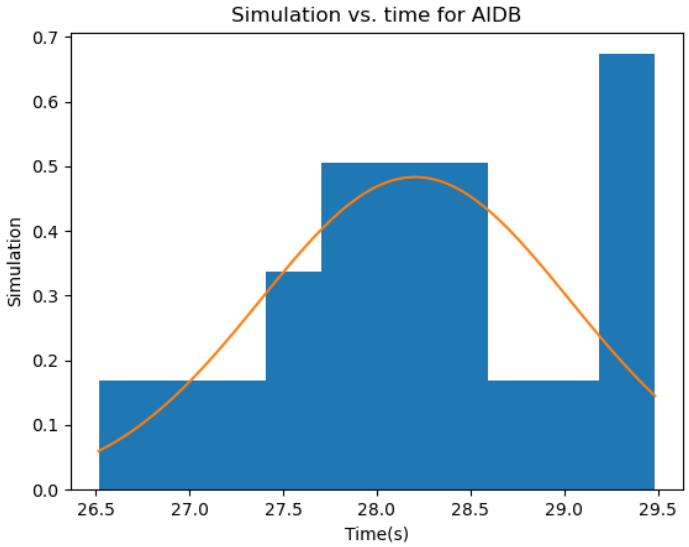



```
sqlite> select * from Mlmappings;
1|frame0195.png|
2|frame0210.png|verbiage
3|frame0225.png|verbiage
4|frame0435.png|verbiage
5|frame0450.png|
6|frame0690.png|
7|frame0705.png|
8|frame0750.png|notation
9|frame0765.png|
10|frame1020.png|
11|frame1035.png|accessory
12|frame1050.png|accessory
13|frame1140.png|accessory
14|frame1155.png|
15|frame1320.png|
16|frame1335.png|compensation
17|frame1350.png|compensation
18|frame1365.png|compensation
```

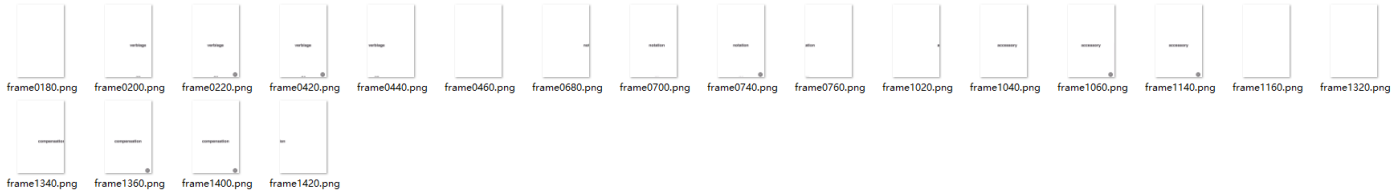
You may wonder how we can record the time a car was within the monitoring range, given that we seemingly only record the entry and exit of the car, with all intermediate frames being discarded in the base table step. This is where the **timestamp** comes into play. As usual, I grouped the same "car number" from the third column but recorded the entry and exit times from the second column. I used the max and min functions to calculate the difference. The frame extraction frequency is our set parameter, which is 15 frames per time, so we perform division accordingly. The time interval is still stored in the third column.

```
sqlite> select * from User_defined_metadata;
1|accessory|8
2|compensation|6
3|notation|1
4|verbiage|16
```

Lastly, I simulated this process 20 times and printed the distribution of the time it took for each run. I fitted the results using a normal distribution. The total time for processing four cars was 31 frames. The error rate is (33-31)/33 = 6.06%, but the time and storage space requirements have been reduced.



For the third method, I use **Approximate Queries** in paper to save more running time and storage space. We set the frequency of video extracting frame to 20 this time.

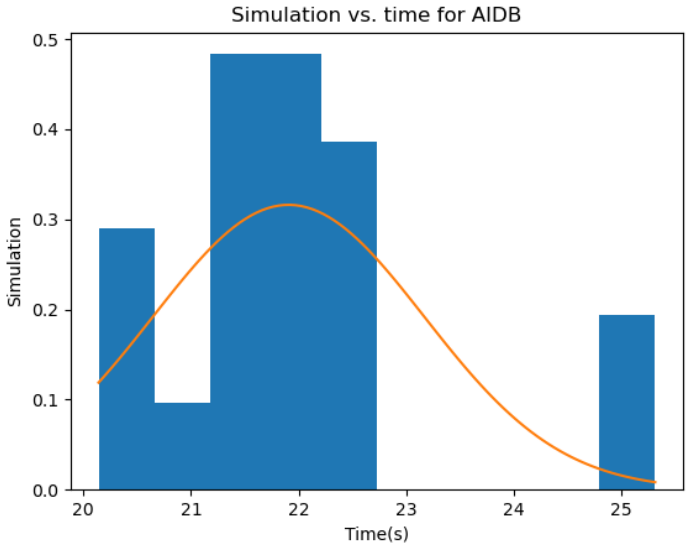



```
sqlite> .schema
CREATE TABLE basetable (id INTEGER PRIMARY KEY, image_name TEXT);
CREATE TABLE Mlmappings (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        image_name TEXT,
        text TEXT
    );
CREATE TABLE sqlite_sequence(name,seq);
CREATE TABLE User_defined_metadata (
        id INTEGER PRIMARY KEY,
        text_grouped TEXT,
        difference INTEGER
    );
```

```
sqlite> select * from Mlmappings;
1|frame0180.png|
2|frame0200.png|verbiage
3|frame0220.png|verbiage
4|frame0420.png|verbiage
5|frame0440.png|
6|frame0460.png|
7|frame0680.png|
8|frame0700.png|
9|frame0740.png|notation
```

```
sqlite> select * from User_defined_metadata
   ...> ;
1|accessory|6
2|compensation|3
3|notation|1
4|verbiage|12
```

Finally, I simulated the same process 20 times and printed the distribution of the time it took for each run. I fitted the results using a normal distribution. The total time for processing four cars was 22 frames. The error rate is (33-22)/33 = 33.3%, but the cost of time and storage space has been further reduced.



For longer-term improvements:

1. The video material used in this project is not an actual traffic camera recording. If real footage is used, OCR recognition will be more challenging. In the beginning, I performed some image processing and included any preprocessing code that may be useful in the future, such as "convert the image," "apply a median filter," and "increase contrast and brightness to grayscale" (Image_preprocess.py & Image_preprocess2.py).
2. The accuracy of **Approximate Queries** needs to be improved, particularly the third method, and the storage space saved needs to be increased. In real traffic monitoring, the fault tolerance rate is low, and the vehicle speed may be higher.
3. The parameter port is not user-friendly. If someone wants to change the frequency of capturing frames, they must modify the source code. To address this issue, I can create software that bundles these parameters together.