# Brief

The Waterview tunnel has installed speed cameras at both ends of the tunnel, these cameras take photos for each car that enters and exits. Using OCR the license plate and timestamp is recorded in a file for each entry/exit of a vehicle. We are required to make a program which generates tickets (with corresponding fine) for vehicles that are speeding in the tunnel.

The target user(s) for the program are the operators of the Waterview Tunnel, in this case the NZTA (New Zealand Transport Authority)

# Key Dates/Timeline

* Week 1: Outcome 1.1 to 1.4
* Week 2: Outcome 2.1 to 2.2
* Week 3:
* Week 4: Outcome 3.1 to 4.1

***Due 7TH APRIL***

# Requirements of program

* List of tickets generated
* Total value of each ticket
* Registration times (Entry, Exit, Speed and fine)
* Warnings of cars which haven’t exited after x amount of minutes

# Design Specifications

* Python 3.8 or above
* Git source control
* PyQT Graphics Library (for gui)
* Try to adhere to PEP 8 when possible (not always possible since using PyQT which is a C++ based library so methods use c++ styling eg. camelCase & PascalCase)

# Planning Notes

* User interface should be graphical
* Files should be in .txt format
* Files should be read from a “./data” folder (relative to project path)
* User should be able to select input and output file path
* Program should be able to identify a vehicle which the license plate may have not been read correctly (eg. Similar matches)
* User should be able to individually export data for vehicles which have not left or license plate does not 100% match as of entry and exit
* There should be a file where fines can be modified eg. .json, .csv or .txt
* Export should be in CSV as it’s a universal file standard used by many other programs

# Potential Difficulties

\* = Out of our control, something we can’t fix unless we had data from both camera separate.

* Incorrect formatting of input data ie. Time then plate
* Incorrect time formatting ie HH:MM vs HH:MM:SS – However I did ask the stakeholder (Mr Hook) and he emailed a reply and said that the format of data will be HH:MM:SS
* Vehicles don’t have a time associated with them
* \*Vehicles may slip through at entry time when camera dumps data hence they only get recorded as exiting, but since we don’t know what camera the data came from it looks like they didn’t exit
* Daylight savings
* \*Plates that OCR can’t recognize hence will not match up in entry or exit data.
* Invalid input data eg. Rego > 6 chars

# Input Data (planned)

|  |  |  |
| --- | --- | --- |
| Variable Name | Data Type | Inputted by |
| path\_file\_in | String | File Explorer Entry Dialog (txt or csv) |
| path\_file\_out | String | File Explorer Entry Dialog (txt or csv) |
| Fine\_ranges | Integer (appended to a list) | Number Picker Entry |
| Fines | Integer (appended to a list) | Number Picker Entry |
| Speed\_limit | Integer | Number Picker Entry |
| Tunnel\_length | Integer | Number Picker Entry |

# Input Data (Actual)

fileName, \_ = QFileDialog.getOpenFileName(

self,

"Open File",

"",

"Comma Separated Values (\*.csv);;Text Files (\*.txt)"

)

        speedLimit = self.speedSBox.value()

        exitThreshold = self.exitSBox.value()

        entryDist = self.entryDistSBox.value()

        tLen = self.tLengthSBox.value()

        exitDist = self.exitDistSBox.value()

# Output Data (planned)

|  |  |  |
| --- | --- | --- |
| Variable Name | Data Type | Outputted by |
| generated tickets | 2D List (string) | Exported as a csv file |
| generated errors | 2D List (string) | Exported as a csv file |

# Output Data (actual)

        self.fines = []

        self.errors = []

# Stored/Indexed Data

|  |  |
| --- | --- |
| Variable Name | Data Type |
| errors | 2D List (string) |
| buffer | 2D List (string) |
| fine\_ranges | 2D List (integer) |
| fines | List (integer) |

fine\_ranges = [[0,10], [11,15],[16,20],[21,25],[26,31],[31,35],[36,40],[41,45]]

fines = [30, 80, 120, 170, 230, 300, 400, 510, 630]

# has to be predefined not global

errors = []

buffer = []

# Modules in program

For calculation purposes

## Read Files:

At the start of the program we use the ReadFile function to read a file and export it to a list which can hold all this data in memory. The function splits each line of data read into 2 pieces of data which will get appended to a list, then that list appended to the main list holding all read data. This is good as it means we don’t have to split later and hold a long string with useless whitespace ie. “ABC123 11:01:01”.

def ReadFile(file\_path):

    global reg\_time

    reg\_time = []

    errors.clear()

    file\_read = open(file\_path, "r", encoding="ascii", errors="ignore")

    if file\_path.endswith("txt"):

        for line in file\_read.readlines():

            line\_data = line.split()

            CheckPlate(line\_data)

    elif file\_path.endswith("csv"):

        for line in file\_read.readlines():

            line\_data = line.split(",")

            CheckPlate(line\_data)

        for i, item in enumerate(reg\_time):

            item[1] = item[1].strip("\n") # strip the newline register of the end of the string

    file\_read.close()

**Notes:**

In the function we essentially just check for the filetype and depending on the file type the way we handle separating data is different. For csv we have to split by the comma and then strip the newline register at the end (\n). For a txt file we can simply split by whitespace, whether that is a space or a tab. When the file is read it is read with ascii encoding and any characters that aren’t recognized get ignored, this means we can read something with possibly corrupt data eg. (ÿYGA825).

## Check license plates

The aforementioned function (ReadFiles) uses the CheckPlate function to verify if the data read is valid, if it isn’t the data will get added to the errors list with a corresponding error message. This is a easy way to weed out invalid data which could cause problems later ie. Vehicles without a license plate.

def CheckPlate(file\_data):

    if len(file\_data) == 1:

        file\_data.append(None)

        file\_data.append(None)

        file\_data.append("No time associated with vehicle")

        errors.append(file\_data)

    elif len(file\_data[0]) <= 6 and file\_data[0].isalnum() == True:

            reg\_time.append(file\_data)

    else:

        file\_data.pop(1)

        file\_data.append(None)

        file\_data.append(None)

        file\_data.append("Invalid License Plate")

        errors.append(file\_data)

**Notes:**

In the function we use this to weed out invalid plates, plates with invalid or missing data. This means we don’t have to worry about errors later on where the index of the list doesn’t exist for something ie. Time In or Out.

## Convert strings to a datetime object

The StrToTime function is possibly the most integral as the program uses it to get the time data provided in the string type, and convert it to a datetime.time object which can later be used for easy calculations. This allows us to work with time much easier.

def StrToTime(time):

    try:

        time = datetime.datetime.strptime(time, "%H:%M:%S").time()

        return time

    except:

        try:

            time = datetime.datetime.strptime(time, "%H:%M").time()

            return time

        except:

            pass

        raise Exception("Time in incorrect format")

        return None

**Notes:**

In the function we use try except statements to let us convert two formats of time that we will accept, the first being HOURS:MINUTES:SECONDS and the second being HOURS:MINUTES. The other format of time, just hours has a too large margin of error and makes it more or less impossible to get a acceptable fine. We use the built in function .strptime() from the datetime library which lets us input a string which will have the time stripped from it, and then a string formatter which lets us decide the time format. If both HH:MM:SS and HH:MM fails we then return no time but raise an error which is displayed to the user and tells them that the time couldn’t be processed due to incorrect formatting.

## Datetime.time object to integer

This function lets us get a datetime object and then returns the time in minutes as an integer. This is useful for speed calculations as we can’t divide datetime.time objects with integers.

def TimeToIntMins(time):

    total\_seconds = int(time.total\_seconds())

    return total\_seconds/60

**Notes:**

In this function we use the built in member function of the datetime.time object .total\_seconds() which returns the total seconds. This gets type casted into an integer. The integer then gets divided by 60, which gives us minutes. This is returned to the calling object.

## Find time in and out

The find time in and out function lets us associate the times in and out of the vehicle. Since we are just given raw OCR data by the camera the program must find what time the vehicle enters and exits. This is as the data isn’t paired ie.

ABC123 11:01:01

ABC123 11:03:55

def FindTimeInOut(reg\_list):

    new\_reg\_list = []

    for i, reg in enumerate(reg\_list):

        plate = reg[0]

        timeIn = reg[1]

        for x in range(i+1, len(reg\_list)):

            if reg\_list[x][0] in plate:

                if timeIn != reg\_list[x][1]:

                    timeOut = reg\_list[x][1]

                    break

                else:

                    timeOut = "00:00:00"

            else:

                timeOut = "00:00:00"

        new\_reg\_list.append([plate, timeIn, timeOut])

    return new\_reg\_list

**Notes:**

In this function we take a list as a parameter we will refer to as A, and then initialize a new list which we will refer to as list B. We then loop through list A and in each iteration, the number plate is stored and time in of vehicle is stored (as we would expect it to be the first occurrence of the vehicle. We then will start a new loop through the list from the current index of list a + 1 (which is the next item in list). In this ‘sub-loop’ we check if the license plate matches, if it is true we then check if the time out isn’t the same as the time in, if that is true we set the time out variable to be the time associated with that occurrence in list a. This sub loop will break and execute the remaining code in the main loop which appends the found data into list b. If the plates or times didn’t match then timeOut would be set to 00:00:00 time and then when the iteration of the list ends, will be appended to list B.

## Remove duplicates

In the find time in and out function, sine this loops through the list twice we actually get two occurrences of in and out times. We use the remove duplicates function to remove duplicates (as in the name, ditto). While yes we could have used other methods ie. Only loop through the lists length/2, this would cause other errors in the find time in and out function, and this was the most versatile solution. This function is also better than looping through the length of list /2 as if a vehicle shows up multiple times those other occurrences just get ignored.

def RemoveDuplicates(list):

    new\_list = list

    for i, item in enumerate(new\_list):

        for x in range(i+1, len(new\_list)):

            if new\_list[x][0] in item[0]:

                new\_list.pop(x)

                break

    return new\_list

**Notes:**

In this function we take a list as a parameter which gets copied into a new list variable. We then loop through the list and have a sub-loop in that loop which start from the current index + 1, which is the next element in the list (this is all iterating through the copied list). In the sub loop we will just check if the license plate matches, if it does we will delete that item from the list, break out of the sub loop and then continue in the main loop to the next index and start the sub loop again. This happens until the list has no more items to iterate over (which means that there are no duplicates). We then return this new list. This is good as it means we never actually modify the original data which can bring up other errors due to list indexes and data scopes.

## Calculate Duration

The calculate duration function, is as the name says used for calculating the duration between two times. This is useful in the program as to calculate average speed we need to know the duration that the vehicle was in the tunnel as v = d/t.

def CalculateDuration(time\_in, time\_out) > datetime.time:   # return type

    min\_date = datetime.date.min

    if time\_out < time\_in:  # if the time out is before the time in, then add 60 minutes (daylight savings)

        new\_time\_out = datetime.datetime.combine(min\_date, time\_out) + datetime.timedelta(hours=1)

        return datetime.datetime.combine(min\_date, new\_time\_out.time()) - datetime.datetime.combine(min\_date, time\_in)

    else:

        return datetime.datetime.combine(min\_date, time\_out) - datetime.datetime.combine(min\_date, time\_in)

**Notes:**

In the function we store the minimum date because this is used in the combine function. The program uses the minimum date because it makes more sense rather than getting the current date as that could cause errors at different processing times. Then we check if the vehicle exited before it left. If it did then we can assume that daylight savings has taken effect during the vehicles time in the tunnel. For the duration calculating we use the .combine function which lets us get timedelta objects to subtract the times. The only difference between the daylight savings calculation and the standard calculation is that we add one hour to the result of the daylight savings result so all further calculations occur as necessary.

## Calculate Average Speed

The calculate average speed function is used for exactly what the name says. We need the calculated speed to that we can see if people were speeding. We use the formula of v = d/t (velocity is equal to distance over time) distance being the value inputted by the user in the GUI (default is 2690m).

def CalculateAvgSpeed(duration\_mins, tunnel\_len):

    return (tunnel\_len / 1000) / (duration\_mins / 60)

**Notes:**

In the function it’s pretty basic. We get the length of the tunnel and divide it by 1000, this gives us the distance of the tunnel in kilometers (as we want output to be km/h not m/s). That is then divided by the duration that the vehicle was in the tunnel. The duration is divided by 60 to give us total hours the vehicle was in the tunnel (again to calculate km/h not m/s).

## Calculate Fines

The calculate fines function lets us see if the vehicle was going over speed at all, and if it was how much it should be charged for going overspeed.

def CalculateFines(speed, speed\_limit):

    over = speed - speed\_limit

    if over <= 0:

        return None

    elif over > fine\_ranges[len(fine\_ranges)-1][0]:

        return fines[len(fines)-1]    # return the last fine in the list

    else:

        for i, x in reversed(list(enumerate(fine\_ranges))):

            if over >= fine\_ranges[i][0]:

                return fines[i]

                break

**Notes:**

In the function we take the speed the vehicle was going and the speed limit inputted by user via gui as parameters. We then calculate how much over the vehicle was going. This is done by subtracting the speed limit from the current speed ie;

Speed = 120km/h

Speed Limit = 80km/h

Overspeed = 120 – 80

= 40 (km/h)

If the vehicle was not going overspeed (less than or equal to 0) then we can break out of the function immediately and return a None type which tells the program (in other functions) the vehicle didn’t speed.

However if the vehicle did happen to be going overspeed then we will first check if it was going over the last fine range from the fine\_ranges list. This also happens to be the cutoff point where the fine value doesn’t increase anymore. If that is true we will return the corresponding fine from the fines list. If that statement returned false we would then goto the else statement where the program will iterate through the fine\_ranges list backwards. This is because it makes more sense to iterate backwards because if we lets say iterate forwards (20-30,31-40,41-50) and the vehicle was going 40 over, that means we would return true for 20-30 and 31-40. This means that in code the program would have to wait until it returns false AFTER a true statement and then know that the last true statement was correct. However by iterating backwards we can find the first fine range which returns true and then we know that it’s correct since the largest one would take precedence vs smaller ones. This basically removes the number of iterations/operations required in most cases, unless in some cases where a vehicle would only be going in the lower end of the ranges ie. (10-20,21-30)

\*\* NOTE: The fine ranges provided in the notes section are not accurate, just arbitrary for explanation purposes.

## Generate Fines

The Generate Fine function links all the calculation functions together to generate duration, speed and fines if applicable. This links all the data to the license plate in either the warnings or main fines list.

def GenerateFines(rtimes\_list\_in, speed\_limit, exit\_threshold, tunnel\_len):

    rds\_list\_out = []   # reg, duration, speed

    for i, item in enumerate(rtimes\_list\_in):

        duration = CalculateDuration(StrToTime(item[1]), StrToTime(item[2]))

        speed = CalculateAvgSpeed(TimeToIntMins(duration), tunnel\_len)

        fine = CalculateFines(speed, speed\_limit)

        if speed < 0:

            errors.append(item)

            errors[len(errors)-1].append("Vehicle did not exit")

        elif TimeToIntMins(duration) >= exit\_threshold:

            errors.append(item)

            errors[len(errors)-1].append("Vehicle did not leave after {} minute exit threshold!".format(exit\_threshold))

        if fine == None:

            continue

        new\_list = item

        new\_list.append(str(duration))

        new\_list.append(round(speed, 2))

        new\_list.append(fine)

        rds\_list\_out.append(new\_list)

    return rds\_list\_out

**Notes:**

We first initialize a list which will store all data that must be returned. We begin to iterate through one of the lists required in function parameters.

In the loop we calculate duration, via the CalculateDuration function. We pass the time in and out of the vehicle as datetime.time types via the StrToTime function.

Once duration is calculated, we calculate the vehicle speed using CalculateAvgSpeed the duration (which was just calculated) converted to an integer via the TimeToIntMins function and also passing the tunnel length (which was also another function parameter of generate fines).

After the speed is calculated we calculate the fine, the CalculateFines function will take the speed we just calculated and the speed limit which was a parameter of the Generate Fines function.

Once all those are calculated and assigned to their respective variables, we can then see if any errors occurred which means the plate should be appended to the errors list.

The first check is if the speed is less than 0, which means the vehicle was going of a negative speed which is impossible. That would mean the entry or exit time of the vehicle was 00:00:00 which was actually assigned by us previously in the program to be used as an identifier.

If this returns true we append the vehicle to the errors list with the error message (“Vehicle did not exit”). If this check returned false we move to the next check. Which is checking if the vehicle took longer than the threshold time before warning (inputted by user in GUI). If this returns true the program will append the vehicle to the errors list with the error (“Vehicle did not leave after {exit\_threshold} minute exit threshold”). Exit threshold is a also a parameter of the generate fines function.

If that returned false, we will continue the loop where another check will occur, but not part of the first if statement. This check if for if the fine is of a None type. If it is then we will just skip to the next iteration of the loop, this is because the vehicle doesn’t need to added to the fines list if it didn’t do anything wrong. If that returns false we will execute the remaining code in the loop which adds the vehicle & calculated data to the fines list.

# Modules for external usage

(GUI Linker functions, does not do any calculations)

## Generate Function

The generate function is one of the main functions used by the GUI. It takes all the possible variables inputted by the user and then calls the correct functions to generate fine & error data.

def Generate(file\_path\_in, speed\_limit, exit\_threshold, tunnel\_len):

        try:

            ReadFile(file\_path\_in)

        except Exception as exc:

            return exc

        try:

            global buffer   # defined as global since we are assigning it in the function (read LEGB scope)

            buffer = GenerateFines(RemoveDuplicates(FindTimeInOut(reg\_time)), speed\_limit, exit\_threshold, tunnel\_len)

            return None

        except Exception as exc:

            return exc

**Notes:**

The program use try excepts so the program doesn’t crash. First we read the file using the file path provided in the function parameter. If that fails we return the exception, this will show an error message box to the user with the error the program returned. This is only for rare error cases as it means we don’t have to cater for every possible error.

The same thing is done with the next try except statement however we are calculating the fines. We create a global variable buffer, however you may notice that buffer already exists in the program, we have to declare it as a global as part of the python LEGB scope rules. We then assign the buffer variable to the ouput of the GenerateFines function.

If there is an exception we will again return the exception to the user via a message box. If there is no exception we return None so the program can verify that everything was calculated correctly.

These try excepts are a last barrier of defence to prevent program crashes.

## File writing

The next two modules are to write files.

### Write text file

def WriteTxt(file\_path, data\_list):

    write\_file = open(file\_path, "w+")

    write\_file.write("Plate\t Time In\t Time Out\t Duration\t Speed\t\t Fine\n")

    for i, item in enumerate(data\_list):

        str\_out = "{}\t {}\t {}\t {}\t {:.2f}km/h\t ${}\n".format(item[0], item[1], item[2], item[3], item[4], item[5])

        write\_file.write(str\_out)

**Notes:**

In the function we open the file with the file path as per function parameter. The program will open the path in w+ mode which means if the file doesn’t exist it will create it. We will then write a header to the file, since this is .txt we separate by tabs.

Then we loop through the list provided in function parameters, we will write the data in the expected format of Plate, time in, time out, duration, speed and fine.

### Write CSV

def WriteCsv(file\_path, data\_list):

    write\_file = open(file\_path, "w+")

    write\_file.write("Plate,Time In,Time Out,Duration,Speed,Fine\n")

    for i, item in enumerate(data\_list):

        new\_item = [str(x) for x in item]

        str\_out = ",".join(new\_item)+"\n"

        write\_file.write(str\_out)

**Notes:**

The write csv function does the same as the .txt with opening the file and writing the header, however instead of using tabs to separate we use commas (,) and add a newline register at the end of each line.

In the loop we join each item of the vehicle data sub-list to a comma then write it. This means we add each piece of information to the string that will be written chunk by chunk and separating by commas.

### Write Errors

def WriteErrors(file\_path, data):

    write\_file = open(file\_path, "w+")

    write\_file.write("Plate,Time In,Time Out,Error\n")

    for i, item in enumerate(data):

        new\_item = [str(x) for x in item]

        str\_out = ",".join(new\_item)+"\n"

        write\_file.write(str\_out)

**Notes:**

The write errors function is exactly the same as writecsv however the header is changed to the error list format of Plate, Time In, Time Out and error message. We only need to write errors as a csv as we don’t allow the user to save errors in .txt formats.

# Testing data

## Rego & Time Test Data (expected)

|  |  |
| --- | --- |
| Input | Expected Output |
| HWS987  DRH396  HB606 10:28  RS437 10:29  FX961 10:29  KHH28 10:30  HB606 10:31  RS437 10:32 | HWS987 No Time Inputted  DRH396 No Time Inputted  HB606 10:28 Nothing  RS437 10:29 Nothing  FX961 10:29 Didn’t Leave  KHH28 10:30 Didn’t Leave  HB606 10:31 Nothing  RS437 10:32 Nothing |
| ERRRORRORORORORORROROR 11:06:27  asodiasosidjaosdjaoisdj 12:07:29  oasidjaoisdjaosidj 12:08:30 | ERRRORRORORORORORROROR Invalid Plate  asodiasosidjaosdjaoisdj Invalid Plate  oasidjaoisdjaosidj Invalid Plate |

## Rego & Time Testing Data (v1 no gui)

|  |  |
| --- | --- |
| Input | Output? |
| HWS987  DRH396  HB606 10:28  RS437 10:29  FX961 10:29  KHH28 10:30  HB606 10:31  RS437 10:32 | Pass, No fines generated; Errors were only:  HWS987 No time associated with vehicle  DRH396 No time associated with vehicle  FX961 10:29 Vehicle didn’t exit tunnel  KHH28 10:30 Vehicle didn’t exit tunnel |
| ERRRORRORORORORORROROR 11:06:27  asodiasosidjaosdjaoisdj 12:07:29  oasidjaoisdjaosidj 12:08:30 | Pass, All generated errors as expected |

## GUI Input Test Data

|  |  |  |
| --- | --- | --- |
| Input Variable | Data inputted | Allowed? |
| Speed Limit (km/h) | EEE  401  999  150  1 | NOPE – PASS  NOPE – PASS  NOPE – PASS  YES – PASS  YES – PASS |
| Exit threshold (minutes) | 61  5  QQQQQQ | NOPE – PASS  YES – PASS  NOPE – PASS |
| Tunnel Length (meters) | 10000  ABASFF  99999999  1 | YES – PASS  NOPE – PASS  NOPE – PASS  YES – PASS |
| Entry Distance (meters) | 10000  600  ABASFF  99999999  1 | YES – PASS  YES – PASS  NOPE – PASS  NOPE – PASS  YES – PASS |
| Exit distance (meters) | 10000  600  ABASFF  99999999  1 | YES – PASS  YES – PASS  NOPE – PASS  NOPE – PASS  YES – PASS |

# Program operations

Below is a flowchart of what happens after the Generate button has been pressed.

Diagram

Description automatically generated

# GUI User operation flowchart

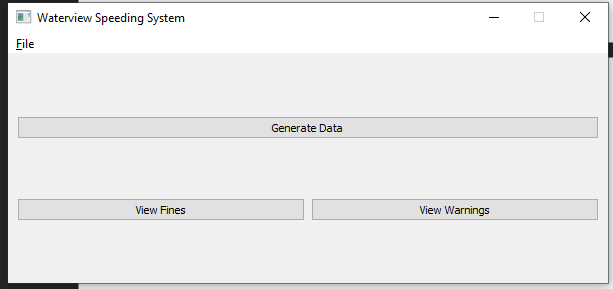
To view bigger photo, right click and save (it is a high DPI image so it won’t pixelate) OR zoom in.

Diagram

Description automatically generated

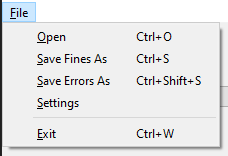
# User Documentation

## The interface

The main UI has 4 main elements.

1. The File Menu
2. Generate Data button
3. View Fines button
4. View Warnings button

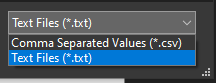
## File menu

The file menu has 5 options:

* Open – With shortcut CTRL+O
  + This opens a file dialog to select a file to read.
* Save Fines As
  + This opens a file dialog to select the output path of the fines generated.
* Save Errors As
  + This opens a file dialog to select the output path of the errors generated.
* Settings
  + This opens the settings window which lets the user change settings about the fine generation.
* Exit
  + Closes the program

## Selecting a file

Click on the file menu and then click open, this will open a file dialog where you can find the location of the .txt or .csv file you want to use to generate data.

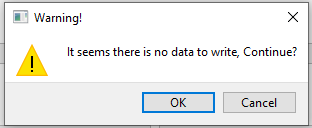
At the bottom corner you can select what type of file you want to select. 

## Saving a file

If you are saving fines goto File and then press the Save Fines as button, this will open a dialog showing where you want to save the file. Once you press save the data generated will be saved to that location

If you are saving errors you can do the same, goto file and then press Save Errors As. However for saving errors you can ONLY export as CSV so beware.

If no data has been generated a warning will popup, however if you feel this is wrong you can still save the file.

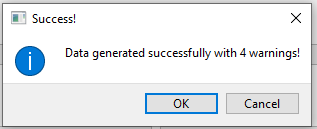


## Generating Data

Once a file has been imported click the generate button.

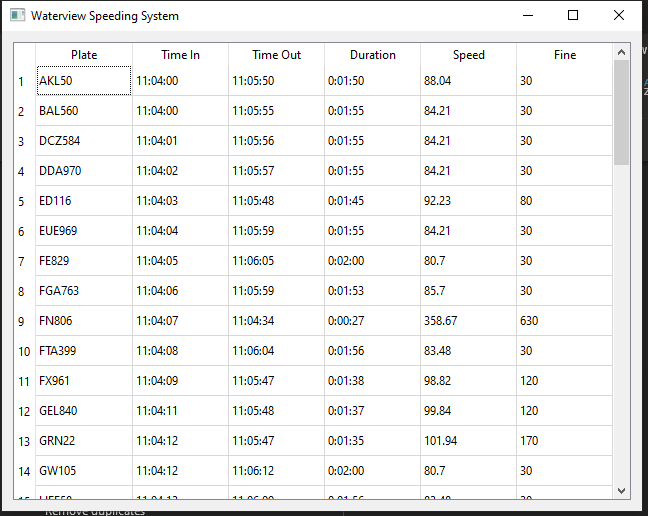
If data has been entered correctly and no errors occurred during the generation process a popup will show saying generation was successful with the Number of warnings/errors.

This button will generate both fines and errors

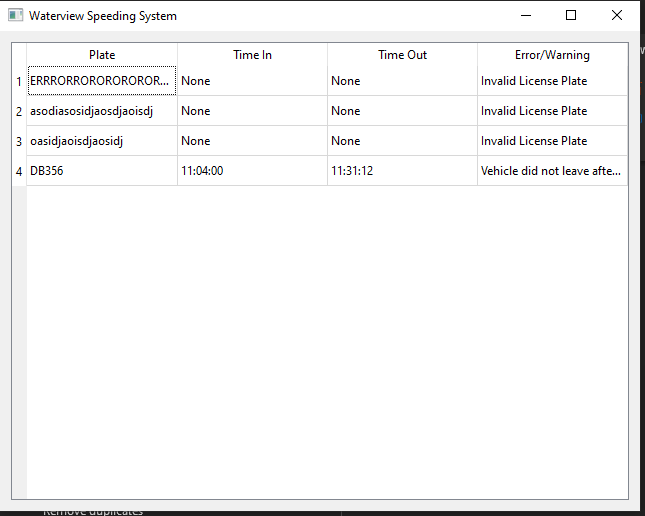


## Viewing Data

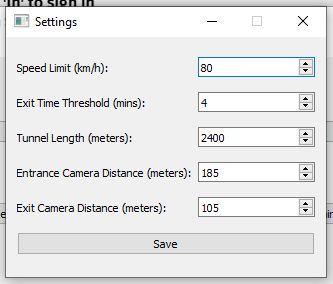
Below the generate button there are two buttons, one to view fines and one to view warnings/errors. Once clicked these will open a new window with a tableview window of the data type selected.



At the top of the table there are heading which correspond to the data shown in each column. At the right there is a scrollbar so you can go down and view more data.



## Settings

To access settings goto file then settings. This will open the settings window. When the settings window shows you will see options to modify the:

* Speed Limit
  + This modifies the speed limit which changes the cutoff before someone gets fined
* Exit threshold
  + This modifies the amount of time that a vehicle can be in the tunnel before it gets added to the warning list
* Tunnel length
  + Length of tunnel, ditto
* Entrance Cam Distance
  + Distance between entrance camera and tunnel entrance
* Exit Cam Distance
  + Distance between exit camera and tunnel exit

These settings are auto populated with default values for the waterview tunnel, should these values ever change they can be modified.

**IT IS IMPORTANT YOU PRESS SAVE OTHERWISE CHANGES WON’T APPLY**

## Data format

Data should be formatted correctly otherwise calculations may not work correctly.

|  |  |
| --- | --- |
| TXT | CSV |
| PLATE TIME  rego HH:MM:SS  Example:  ABC123 11:05:05 | PLATE,TIME  Rego,HH:MM:SS  Example:  ABC123 ,11:05:05 |

Even though the HH:MM:SS format for time is preferred we can still accept HH:MM

|  |  |
| --- | --- |
| TXT | CSV |
| PLATE TIME  rego HH:MM  Example:  ABC123 11:05 | PLATE,TIME  Rego,HH:MM  Example:  ABC123 ,11:05 |

**NOTE: FOR TXT MAKE SURE DATA IS SEPARATED BY SOME SORT OF WHITESPACE, WHETHER TABS OR SPACES.**

## Steps

1. Import a file

2. Adjust Settings (optional)

3. Generate Data

4. View Data