Documentation of

“Detecting advanced lane features”

project

Sandeep Yadav Mattepu, Tony John

February 2020

List of Figures

Contents

Preface

What is this document?

This document documents the project that might be used by oTTo car team to detect signs on the lane for driverless car in *Carlo-Cup* competition.

Who is this document for?

This document assumes that reader has good knowledge in programming languages like C++ and python. It also assumes that user has knowledge in both *ROS* and *OpenCV*.

Other resources

Source code and C++ API documentation of this project are available in the zip file in which this document is present.

Chapter 1

Compiling and running the project

1.1 Compiling the project

Copy the source code(i.e *detect_sign_on_lane*) folder and paste it inside *src* folder of your ros workspace. This project requires other packages to compile. Following are the list of packages that it requires:

- roscpp
- std_msgs
- std_srvs
- cv_bridge
- sensor_msgs
- message_generation
- ottocar_msgs(Custom package created by oTTo car team)

If you do not have any of the above mentioned package(except *ottocar_msgs*) install them by opening the terminal and typing the following command.

```
# rosdep install detect_sign_on_lane
```

For *ottocar_msgs* package, you need to get it from oTTo car team.

Once you have all the package installed in your system it's time to compile the project. Navigate your terminal to ros workspace

folder and enter following command.

```
# catkin_make
```

1.2 Running the project

To run the project you have to enter the command in the terminal in the following format(Make sure *roscore* is running before running following command):

```
# rosrun detect_sign_on_lane detect_sign_on_lane -m [Location of svm file]
  -d [Debug mode]
```

- **Location of svm file :** This file is in /path/to/your/ros_workspace/devel/lib/detect_sign_on_lane folder and file is svm_data.dat
- **Debug mode :** Boolean value instructing whether to run in debug mode or not. If you want to visualize the results of detection then set this value to true otherwise set it to false. When set to true there will be another topic with name */detect_sign_on_lane/debu-g_result_image* which will publish the top view of lane with detection results drawn on them.

Example : If you wanted to run the project in non-debug mode then the command would be

```
# rosrun detect_sign_on_lane detect_sign_on_lane -m ~/ros_ws/devel/lib/
  detect_sign_on_lane/svm_data.dat -d false
```

Chapter 2

Detection of single contour signs

2.1 Introduction

Several signs like speed limit number, speed limit end, pedestrian island and turn signs are single contour. Detection of these signs are somewhat simpler compared to multi contour signs.

2.2 Detection process

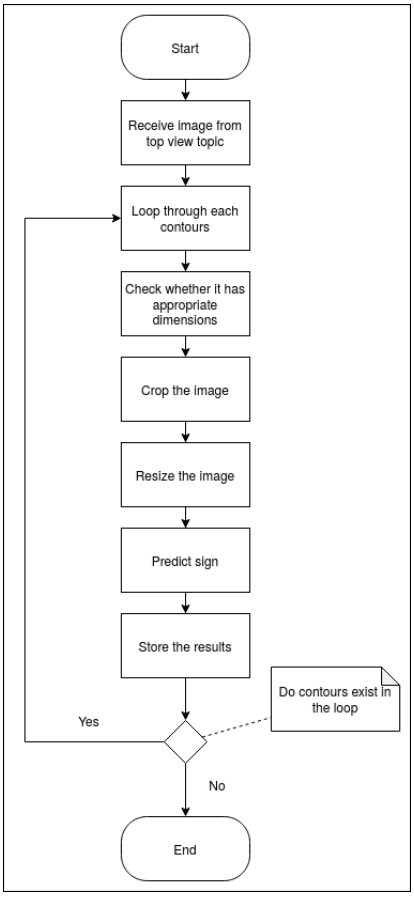


Figure 2.1: Flow chart of detection of single contour sign

2.2.1 Receive image from top view topic

The code in *DetectSignsOnLane::init()* function registers to topic which has top view of the lane.

2.2.2 Loop through each contours

The code in *DetectSignsOnLane::detectSignsFromRawImage()* function loops through the contours of the image in search of the lane signs.

2.2.3 Check whether it has appropriate dimensions

Each contour is checked for their dimensions in different functions based on the sign we are looking for and also each sign has different dimensions. The functions in which dimensions of contour is checked and the constants of dimensions on which they are checked against are mentioned below.

Sign Type	Function name
Number	*DetectSignsOnLane::isContourANumber()*
Turn left/right	*DetectSignsOnLane::isContourATurnSign()*
End of speed limit	*DetectSignsOnLane::isContourASpeedEnd()*
Pedestrian island	*DetectSignsOnLane::isContourAPedestrianIsland()*

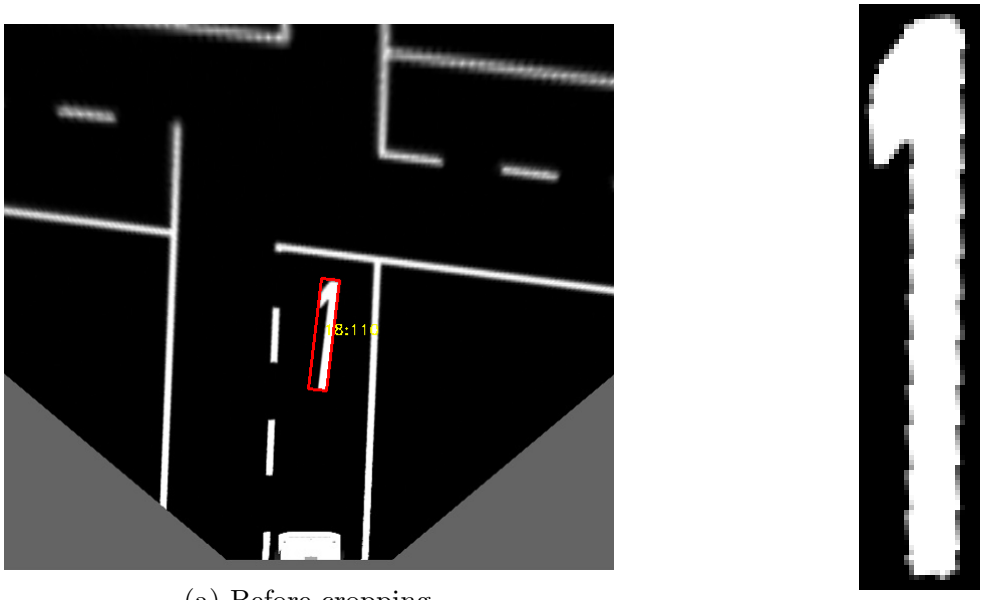
Table 2.1: Different functions where different signs are checked

Sign Type	Constants on which contours are checked against
Number	NUMBER_SIGN_RECT_MIN_WIDTH, NUMBER_SIGN_RECT_MAX_WIDTH NUMBER_SIGN_RECT_MIN_HEIGHT, NUMBER_SIGN_RECT_MAX_HEIGHT
Turn left/right	ARROW_SIGN_RECT_MIN_WIDTH, ARROW_SIGN_RECT_MAX_WIDTH ARROW_SIGN_RECT_MIN_HEIGHT, ARROW_SIGN_RECT_MAX_HEIGHT
End of speed limit	SPEEDEND_SIGN_RECT_MIN_WIDTH, SPEEDEND_SIGN_RECT_MAX_WIDTH SPEEDEND_SIGN_RECT_MIN_HEIGHT, SPEEDEND_SIGN_RECT_MAX_HEIGHT
Pedestrian island	PEDESTRIAN_ISLAND_MIN_WIDTH, PEDESTRIAN_ISLAND_MAX_WIDTH PEDESTRIAN_ISLAND_MIN_HEIGHT, PEDESTRIAN_ISLAND_MAX_HEIGHT

Table 2.2: Constraints on dimensions of different signs

2.2.4 Crop the image

The contours which happens to have appropriate dimensions of any of the single contour sign, then image will be cropped up to the contour. This step is done in *DetectSignsOnLane::cropAndCompressImage()* function.



(a) Before cropping

(b) After cropping

Figure 2.2: Before and after cropping the image for single contour sign

2.2.5 Resize the image

The image which is cropped will be resized to 20x20 pixel size image. This step and previous steps are preprocessing before asking the model to predict the sign.

(a) Before resizing

(b) After resizing

Figure 2.3: Before and after resizing the image for single contour sign

2.2.6 Predict sign

After above preprocessing step, the 20x20 pixel will be sent to the model. The model will predict the sign by outputting a number. *DetectSignsOnLane::predictSign()* is the function that predicts the sign and outputs the number. Details about how to train a model is discussed in upcoming chapter 3. The following is the table that describes the meaning of each number predicted by the model.

Number from model	Sign Type
0	Number sign 0
1	Number sign 1
2	Number sign 2
3	Number sign 3
4	Number sign 4
5	Number sign 5
6	Number sign 6
7	Number sign 7
8	Number sign 8
9	Number sign 9
10	Speed limit ends
11	Turn left
12	Turn right
13	False detections
14	Inverted numbers
15	Pedestrian island

Table 2.3: Different sign and their associated number

2.2.7 Store the results

All the signs that are detected in a single image are stored in a list and the results are published. Details of the publishing the results are discussed in chapter 5.

Chapter 3

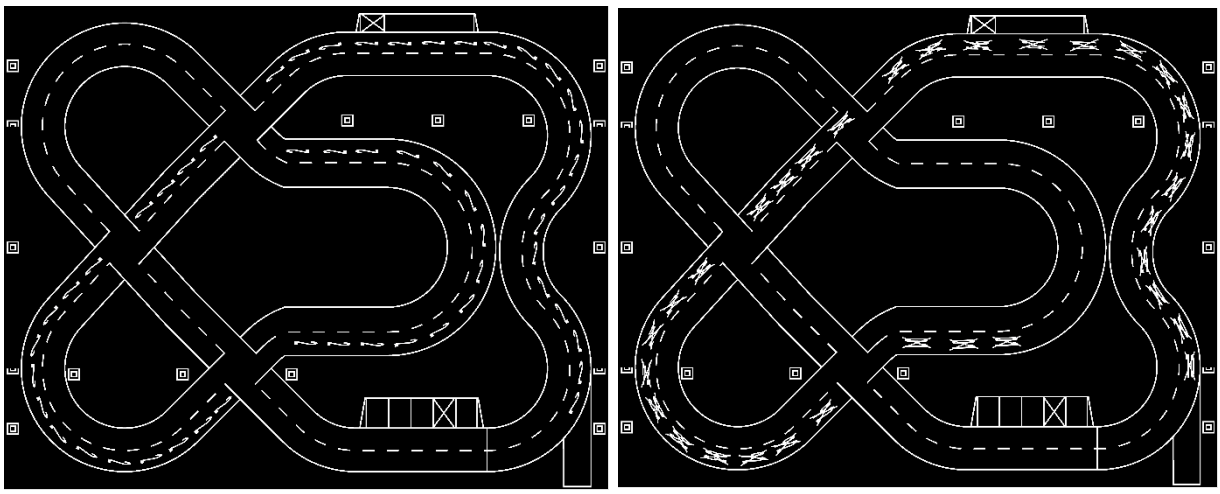
Training the classifier

3.1 Introduction

The model which was used in subsection 2.2.6 was trained with a lot of images. To obtain training images, the **Gazebo** based simulation of OttoCar was used.

3.2 Steps to produce training data

1. Each marking was placed on the track of the simulation numerous times as shown below.



(a) Markings of single number on track (b) Markings of speed end on track

Figure 3.1: Marking signs on track for training data

2. Simulation was run and the top view images generated were saved at 30 frames per second.

3. The images of markings were processed and extracted from the saved top view images as mentioned in the steps of image processing. These 20 x 20-pixel images were saved.

4. Repeat Step 1 to Step 3 for all the markings. There is a total of 13 classes and around 1000 training images were generated for each class.

5. The training images obtained were checked for images not containing a valid marking and invalid images were deleted.

6. Class labels are assigned to the training images.

7. The training images were split into 80% training and 20% testing.

8. HOG features of the images were computed using the parameters:

   - winSize = (20,20)
   - blockSize = (10,10)
   - blockStride = (5,5)
   - cellSize = (10,10)
   - nbins = 9

9. An SVM model using Radial Basis Function as kernel is set up in OpenCV.

10. OpenCV API provides an auto train function to train SVM classifier without specifying the hyper parameters. It does the training multiple times and chooses the optimal hyper parameters. Auto train function was used to train the SVM classifier.

11. Using the test dataset the trained classifier was evaluated. An accuracy of 99% was obtained.

12. The trained model was saved as *svm_data.dat* file.

Chapter 4

Detection of multi contour signs

4.1 Introduction

Signs like zebra crossing and barred area are multi contour. Detection of such signs are discussed in this chapter.

4.2 Zebra crossing detection

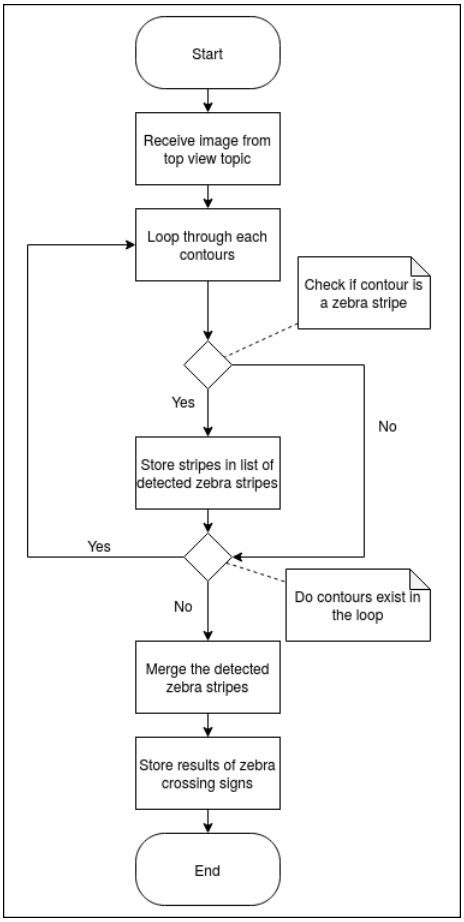


Figure 4.1: Flow chart of detection of zebra crossing sign

Steps **"Receive image from top view topic"** and **" Loop through each contours"** are already discussed in subsection 2.2.1 and subsection 2.2.2

4.2.1 Check if contour is a zebra stripe

From each contour a *cv::RotatedRect* is extracted and it is checked whether it is a zebra stripe or not. This is done in *ZebraCrossing::isRectZebraStripe()* function. Following are the lists of dimensional constants against which *cv::RotatedRect* of the contour is checked for zebra stripe.

- **ZEBRA_CROSSING_MINIMUM_HEIGHT**
- **ZEBRA_CROSSING_MAXIMUM_HEIGHT**
- **ZEBRA_CROSSING_MINIMUM_WIDTH**
- **ZEBRA_CROSSING_MAXIMUM_WIDTH**

4.2.2 Store stripes in list of detected zebra stripes

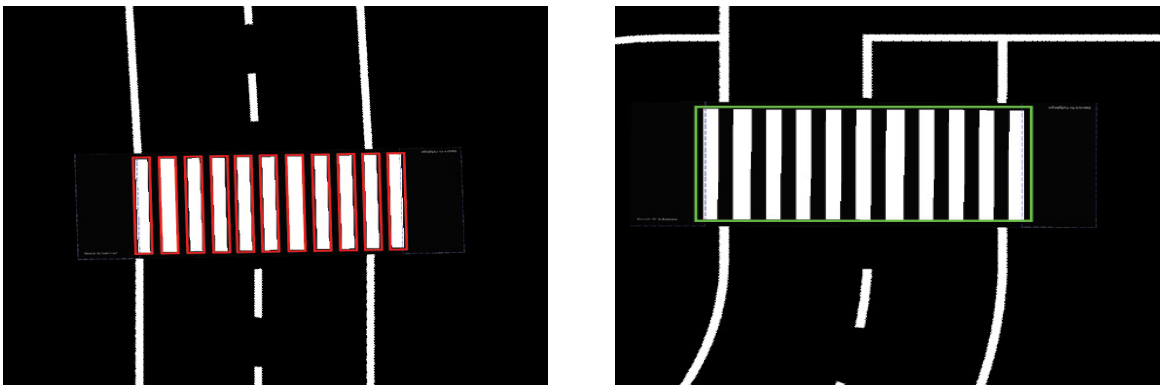
The contours which are detected as zebra stripe in previous step. Those *cv::RotatedRect* are stored in a list for merging them together.

4.2.3 Merge the detected zebra stripes

The list of *cv::RotatedRect* which are candidate zebra stripes are merged together using *ZebraCrossing::clusterStripesForZebraCrossings()* function. This function then outputs the zebra crossing signs on the lane for a given frame.

4.2.4 Store results of zebra crossing signs

The zebra crossing signs which are detected in previous step are stored for publishing them as results. Details about how to publish are discussed in chapter 5



(a) Before merging the zebra stripes (b) After merging the zebra stripes

Figure 4.2: Before and after merging the zebra stripes

4.3 Barred area detection

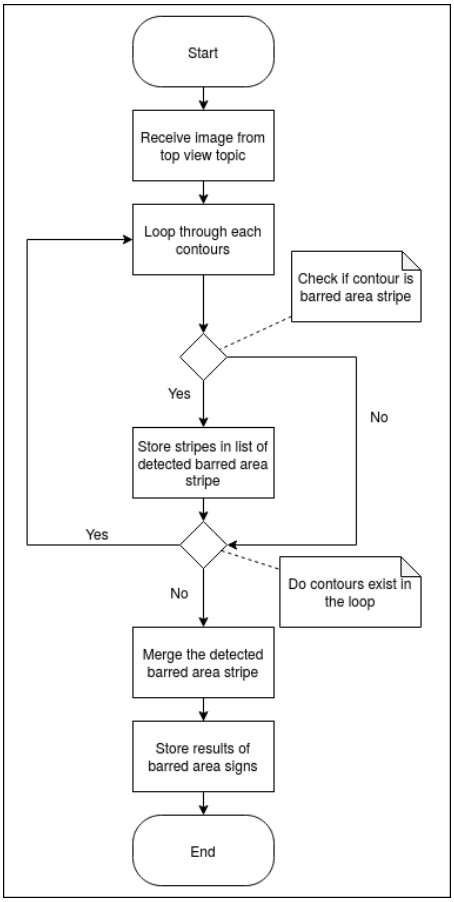


Figure 4.3: Flow chart of detection of barred area sign

Steps **"Receive image from top view topic"** and **" Loop through each contours"** are already discussed in subsection 2.2.1 and subsection 2.2.2

4.3.1 Check if contour is a barred area stripe

Each contour is checked whether it is a barred area stripe or not in *BarredAreaStripe::isBarredAreaStripe()* function. Following are the list of dimensional constants against which the contour is checked for barred area stripe.

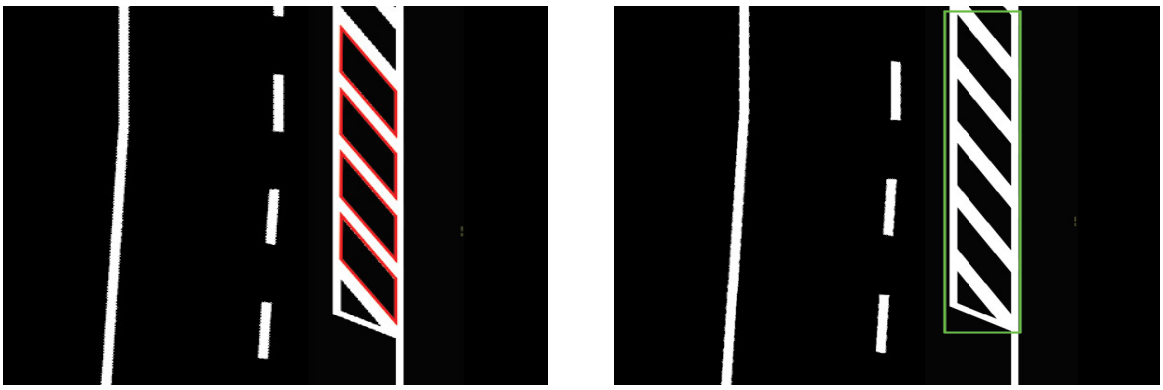
- **SMALLER_SIDE_MINIMUM_LENGTH**
- **SMALLER_SIDE_MAXIMUM_LENGTH**
- **LARGE_SIDE_MINIMUM_LENGTH**
- **LARGE_SIDE_MAXIMUM_LENGTH**
- **MIN_ACUTE_ANGLE**
- **MAX_ACUTE_ANGLE**
- **MIN_OBTUSE_ANGLE**
- **MAX_OBTUSE_ANGLE**

4.3.2 Store stripes in list of detected barred area stripes

The stripes which are detected in previous step are stored in the list of *BarredAreaStripe* type for merging them later.

4.3.3 Merge the detected barred area stripes

The list of *BarredAreaStripe* type are merged together using *BarredArea::clusterStripes()* function. This function outputs the barred area on the lane for a given frame.



(a) Before merging the barred area stripes (b) After merging the barred area stripes

Figure 4.4: Before and after merging the barred area stripes

4.3.4 Store results of barred area signs

The barred area signs which are detected in previous tep are stored for publishing them as results. Details about how to publish are discussed in chapter 5.

Chapter 5

Publish results

5.1 Introduction

In the previously discussed sections(i.e. subsection 2.2.7, subsection 4.2.4, subsection 4.3.4) we have stored the results in a list for publishing them later. This chapter discusses about how results are published.

5.2 Publishing results through topic

The results of detection are published to other nodes through topic Publishing of the results happen in *DetectSignsOnLane::publishResults()* function. The name of the topic will be determined by *DetectSignsOnLane::nameOfResultsTopic* constant value. Debug image topic can also published by *DetectSignsOnLane::publishResultsForDebug()* function under the topic name which will be determined by *DetectSignsOnLane::nameOfDebugResultsTopic* constant value.

5.3 Data structure of results topic

5.3.1 Data structure of *SignOnLaneMsg* msg

Constants that determine the sign type:

- *uint8 SIGNTYPE_SPEED_LIMIT = 0*
- *uint8 SIGNTYPE_SPEED_LIMIT_END = 0*

- *uint8 SIGNTYPE_TURN_LEFT = 0*
- *uint8 SIGNTYPE_TURN_RIGHT = 0*
- *uint8 SIGNTYPE_ZEBRA_CROSSING = 0*
- *uint8 SIGNTYPE_BARRED_AREA = 0*
- *uint8 SIGNTYPE_PEDESTRIAN_ISLAND = 0*

Following are the variables that holds some information of the sign:

signType	This holds value that describes the sign type based on above constants
value	This holds speed limit value. For rest of the signs it's value will be -1
width	Width of the sign
height	Height of the sign
boundingBox	List of 4 points that surrounds the sign

Table 5.1: Data structure of results msg

5.3.2 Data structure of *SignsOnLaneMsg* msg

This message type simply holds array of *SignOnLaneMsg* type

- *SignOnLaneMsg*[] results

Chapter 6

Miscellaneous tools

6.1 Introduction

This project comes with two additional tools to make it easier to change the code and document the code.

6.2 Contour inspector

In Table 2.2, subsection 4.2.1 and subsection 4.3.1 the detection of the signs are dependent on the constant values. These values need to be changed if the camera's properties are changed(like angle, distance, etc.). Whenever camera's properties are changed you can use *inspect_contour* node to inspect contours for their dimension values.

6.2.1 Starting the contour inspector

To start the contour make sure that *roscore* and the node which publishes the topic that needs to be inspected are running. After compiling the project as discussed in section 1.1 enter the following command in the terminal:

```
# rosrun detect_sign_on_lane inspect_contours /topic/name/to/inspect
```

Tool *inspect_contour* provides following functionalities:

- Measure height, width and angle of *cv::RotatedRect* of the contour.

- Distance between the contours.
- Length of each side of barred area stripe and all the angles of the barred area stripe.

6.3 Doxyfile generator

If you modified the code of the project, you may want to regenerate the API code as html. To do this you need to install *doxygen* and have a Doxyfile. Please follow instructions mentioned in http://www.doxygen.nl/manual/install.html to install doxygen. For the Doxyfile, navigate the terminal to the location where DoxyFileGenerator is present. This is present in detect_sign_on_lane/documentation/doxygen/DoxyFileGenerator/bin folder. This binary requires four arguments

- Full path to the project's location in the system.
- Full path where you want html files to save.
- Full path where Doxyfile_Template is present. This is present in detect_sign_on_lane/doc-umentation/doxygen/DoxyFileGenerator/Doxyfile_Template
- Full path where you want DoxyFile to be saved.

Example :

```
# /ros_ws/src/detect_sign_on_lane/documentation/doxygen/DoxyFileGenerator/bin/DoxyFileGenerator /ros_ws/src/detect_sign_on_lane/ /Desktop/Html/
 /ros_ws/src/detect_sign_on_lane/documentation/doxygen/DoxyFileGenerator/Doxyfile_Template /Desktop/
```

Once the Doxyfile is produced, you can now generate the html API documentation by:

```
# doxygen /path/to/new/Doxyfile
```