GSOC 2017 Application
Diptanshu Jamgade:
Python bindings for CCExtractor

About me

Username and Contact Information

Name: Diptanshu Jamgade

University: Indian Institute of Technology, Kharagpur

Email: diptanshuj@gmail.com

Slack Username: *skrill*

GitHub Username: Diptanshu8

Personal Background

Hello, I am Diptanshu Jamgade a fourth year undergraduate student of IIT Kharagpur, India. I am pursing a degree in Electronics and Electrical Engineering. I work on Ubuntu (14.04 LTS and 16.04 LTS) with vim as my primary text editor. Along with vim, I used tmux for multiplexing the work I do. I love vim for its power and flexibility. I'm proficient in C and python.

Contributions to CCExtractor

I started working on CCExtractor in February 2017 and I have been able to make the following merges:

- (**Merged**) 'Fix issue #705 (#721)'.

The Project

Title: Write Python bindings for CCExtractor

The problem and motivation:

CCExtractor is a console application with the sole objective of extracting closed captions from videos. The entire code base encompasses on C/C++ and uses Tesseract-OCR and Libleptonica alongwith curl as its dependencies.

In this proposal, I propose to work on the idea of developing Python bindings for CCExtractor. The main objective of doing so is to make CCExtractor easily accessible and extending it to Python. With growing number of developers in the Python Community, it is really important to extend CCExtractor to Python using Python Bindings.

The major problems faced in doing so are:

1. Presently, CCExtractor is present as a binary or a GUI. The current version of CCExtractor does not allow us to call CCExtractor as a library. So, the first step would be to write some code so that it is possible to use CCExtractor as a binary library.
2. After the library has been developed, it has to be taken care that we can call multiple instances of CCExtractor without causing a Deadlock (race condition). This is another challenge faced. For solving this problem, usage of threads with Data Management using Thread-Specific Data would help us run multiple instances of CCExtractor parallely.
3. After the library has been developed and tested for successfully running multiple instances of CCExtractor together without any errors, then comes the step to extend CCExtractor to Python. For doing so, we can use SWIG (Simplified Wrapper and Interface Generator). It is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages.
4. The documentation of CCExtractor for Python bindings and usage of CCExtractor library has to be drafted from the base.

The plan:

The entire plan for this project revolves around the core concept of converting the present version of CCExtractor to a library which can then be used to write the Python bindings which is the prime objective of this project.

1. For the first problem, i.e., converting the present version of CCExtractor to a library so that it can be used to write the python bindings is a very important step of this project. For solving this problem, I propose a solution as follows:
    a. The library would consist of a function which would initialise the parameters and return a basic structures of the parameters to be parsed to CCExtractor

for processing (similar to init_options() usage in the present code base). The parameter needed for this function is of the type *struct ccx_s_options.*

b. After the parameters have been initialised, then the user can add arguments to set respective parameter values by another function. This would allow to set required parameters for desired output by the user. The user argument parsing could take place one argument at a time.
c. After the parameters have been initialised and the user argument-parsing has been done to set the desired parameter values, then comes the stage of checking if the parameters have a valid value and can they be further processed. This can be done internally using another function and a return value can be used to indicate whether the parsed parameters are valid or do they need any amendments for validation. In this case when the parameters aren't compiled properly the parameters previously flushed will not be flushed away but they would stay in the memory and the user needs to modify them.
d. In the step to compile and check parameters, I would suggest to mandate setting a minimum number of parameters depending on the file type the user intends to provide as input. This would make compiling the parameters a little easy and also, we can ask the user to set the output format or it may be auto-generated. Along with this, a function to reset any particular parameter to its default value could be a great option in case the user wants to reset the parameter (considering the case of new users who are unaware of the correct values for parameters).
e. Once the parameters have been set to desired value and the compilation and validation of the parameters has been successfully completed, then we can proceed further with the actual task of extraction of closed captions from the input file. For doing so I would like to suggest the usage of 3 function.
    i. Start function: This function would be used by the user to start the actual extraction procedure once the parameters have been compiled. This would help the user have flexibility over controlling the initiation of the extraction of the closed captions. An obvious but mandatory sub-procedure here in would be to verify the compilation flag returned by the parameters compilation function so that we can interrupt the process even before the initiation.
    ii. Stop function: Another utility would be to be able stop the processing as and when desired. For this purpose, a stop function would come in handy rather than to wait till the end of the entire process for the finish execution.
    iii. Status function: When an extraction process is being executed it may come handy to check the status of the process and for this purpose on the lines of present structure, a function showing the percentage completion of the process would be really helpful.

NOTE: A clear view on the utility of the above mentioned 3 functions can be understood in context of multi-thread programming. This section is explained in detail later in this document.

The following text box clearly shows a preliminary architecture I would like to suggest for the library header file.

```
#ifndef CCEXTRACTORAPI_H
#define CCEXTRACTORAPI_H

/* Includies */

struct APIParams;

struct APIParams* APIInit();
void APIAddParam(struct APIParams* params, char* arg);
int APICompileParams(struct APIParams* params, char** output);
void APISetOutput(struct APIParams* params, char** output);
void APIStart(struct APIParams* params);
void APIStop(struct APIParams* params);
double APIGetPercents(struct APIParams* params);

#endif //CCEXTRACTORAPI_H
```

Architectural overview of the library

The following section contains small description regarding the usage and parameter detailing for every function shown in the proposed header file.

i. Struct APIParams:
   a. This *structure* would be analogues to the *struct ccx_s_options* which would be initialized and returned by the *APIInit()* function. The elements of this *structure* are then modified depending on the arguments the user passes for processing.
   b. Considering the scope of extension to multi-threading (described later), the instance of this type would be localized to every thread; the return value from *APIInit()* would be added to the thread specific data area so that the threads can run independently without the problem of over writing or loss of data.

ii. Struct APIParams* APIInit():
   a. The *api_init_options* function initialises an instance of the *struct ccx_s_options* and returns the instance. Over in the main function call loop, a pointer of this type is needed to catch the returned value.

```
struct ccx_s_options ccx_options;

struct ccx_s_options* api_init_options(){

   init_options(&ccx_options);

   return (&ccx_options);

}
```

iii. void APIAddParam(struct APIParams* params, char* arg)

a. The *api_add_param* function has been defined with the prime objective of parsing user arguments to modify respective parameters in the memory location pointed by *api_options*.

```
void api_add_param(struct ccx_s_options* api_options,char* arg){

api_options->myarguments = realloc(api_options->myarguments, (api_options->argument_count+1) * sizeof *api_options->myarguments);

api_options->myarguments[api_options->argument_count] = malloc(strlen(arg)+1);

strcpy(api_options->myarguments[api_options->argument_count], arg);

api_options->argument_count++;

}
```

b. For keeping a track of the total number of arguments parsed and also for storing a list of all the parsed arguments to the CCextractorapi, a modification has been made to the *struct ccx_s_options* structure. I have added *char** myargument* and *int argument_count* to the structure. With the help of these options, storing a list of all the arguments parsed by the user becomes possible. Also, it helps in future compilation of parameters.

iv. int APICompileParams(struct APIParams* params, char** output)

```
int compile_params(struct ccx_s_options *api_options,int argc){
        int ret = parse_parameters (api_options, argc, api_options->myarguments);
  if (ret == EXIT_NO_INPUT_FILES)
        {
                print_usage ();
                fatal (EXIT_NO_INPUT_FILES, "(This help screen was shown because there were no input files)\n");
        }
        else if (ret == EXIT_WITH_HELP)
                return EXIT_OK;
else if (ret != EXIT_OK)
                exit(ret);
  return EXIT_OK;
}
```

a. Once the arguments have been parsed properly, then comes the stage to compile parameters and modify respective values in the thread specific structure (*struct* APIParams*) defined in the *init* stage to further initiate the processing using CCExtractor.
b. The definition of the *compile_params* as shown above is a very trivial definition. We can extend this to implement more functionality so as to avoid any collision of the arguments parsed by the user.

v. void APIStart(struct APIParams* params)
a. This function would be similar to the main function in the present version of *CCExtractor.c* except the parameter parsing part.
b. The user may use this function to initiate the execution of CCExtractor on the parsed arguments.
c. The main usage of this function comes in cases of multi-threading (discussed later).
d. The *api_start* function could be found here.

NOTE: A very basic version of the library could be found here (till line number 495).

2. The next important task is that the user may require to call multiple instances of CCExtractor and thus, an extension of the library developed in the above step needs to be extended using threading [*pthread.h; GNU/Linux implements the POSIX standard thread API (known as pthreads); for windows based systems a wrapper called pthreads-win32 has been developed*]. A lot of things are needed to be taken care while doing so and some of the major points have been discussed below:
a. Thread creation:
i. Thread creation is one of the many things that are to be taken care when a thread is to be initiated. The most important step required is to have a unique thread id returned to the user so that the user can access the thread running status. Also, the possibilities to stop any particular thread of access any other thread (dependent) data in the thread specific data (space) are in need of the thread id as a vital parameter.
ii. Each thread in a process is identified by a thread ID. When referring to thread IDs in C or C++ programs, use the type *pthread_t*.
iii. The thread creation can take place at one of the two points based on the present architecture of the library being proposed to be developed.
1. Either we create a thread at the initialization step, thereby keeping all the variables in the context of the main function (basically those variables whose values are dependent to the arguments passed by the user) localized to a particular thread, thereby implementing a thread specific data variables.
2. Or we can create a thread after the user calls the start function of the library wherein we pass the variables in

context to that particular thread as *thread_arguments* to the create thread method.

I would propose for the first solution over the other. The second solution is not up to the mark. Suppose, we choose the second approach to create threads then the case where in a user first feeds in arguments on the basis of a particular file cannot create another thread to run CCExtractor on another video without having run CCExtractor on the first video or having lost the arguments to the first video. For instance, when the user loads the arguments for the second video file even without having started the thread for the first video, then the parameters for the first video would be over-written thereby causing a failure to run CCExtractor for the first video successfully.

b. Passing data to the threads:
    i. The thread argument provides a convenient method of passing data to threads. Because the type of the argument is void*, using the thread argument to pass a pointer to some structure of data is a more efficient approach and it goes with our present workflow. One commonly used technique is to define a structure for each thread function, which contains the "parameters" that the thread function expects.
    ii. Thus in our case if we pass an instance of *struct ccx_s_options* to every thread as a thread attribute, then that could be sufficient to initialise the attributes of a thread to an initial value. Later on, depending on the arguments which a user wants to feed for a particular thread, the user may need to provide the thread id along with arguments to compile the arguments accordingly. This would let the user set arguments for any number of CCExtractor threads and then run the threads in any order rather than just in First-Cum-First-Execution order.
    iii. Now, the compiled arguments in this case would be specific to the thread and cannot be accessed by any other thread. This is how I propose to implement the *thread specific data area.* In case of thread specific data, a new key is created for every data item specific to a thread. All the thread-specific data items are of the type *void** and each is referenced by a unique key.
    iv. After the thread-specific-data keys are generated, then each thread can set/get its thread specific data pertaining to a particular key using *pthread_setspecific/pthread_getspecific*.
c. Thread cancelation:
    i. Under normal circumstances, a thread terminates when it exits normally, either by returning from its thread function or by calling

pthread_exit. However, it is possible for a thread to request that another thread terminate. This is called *canceling a thread.*

ii. To cancel a thread, call *pthread_cancel*, passing the thread ID of the thread to be cancelled. This can be used in the stop function defined in the library description section above.

3. Python Binding Development:
    a. After the library development has been successfully implemented using multi-threading to run multiple instances of CCExtractor in parallel, then comes the main objective of this project, i.e, to develop python bindings for the developed library.
    b. The bindings can be developed either manually or using some other binding library which generates the python bindings.
    c. I would like to propose using SWIG (Simplified Wrapper and Integrator Generator) for generating python bindings.
    d. SWIG is capable of wrapping all of ISO C99 for creating python bindings.
    e. Thus, on creating a proper source and header file for the CCExtractor library, a basic interface file for wrapping the library would be sufficient to generate the Python bindings. This interface file is needed by SWIG to generate the python bindings. In addition, SWIG allows the possibilities to manually add functions to the generated python bindings which can help in development debugging as well as for the overall implementation of some additional features specifically for python developers.
4. Documentation development:
    a. At present, the documentation in context of the functions and definitions made in the library and the python bindings is at the base level.
    b. So once the development of the library and python bindings has reached a particular level, then the documentation of the development made so far would be really mandatory for further development.
    c. So, I propose to work on the documentation of the library as well as python bindings as the development progresses so that the documentation development goes in synchronization with the development of the library as well as python bindings.
    d. The documentation regarding the developed library and developed python module could be either embedded and distributed with the source or it could be hosted on the website. In case of hosting the documentation on the website, I would require support from the organization hosting the documentation on the website.

Architectural clarifications from the point of view of multi-threading:

- The proposed structure of the library in the previous sections are consistent with running a single instance of CCExtractor at a time. However, for running various instances of CCExtractor in threads, modifications need to be done with the present

architecture. The following points give an outline on the modifications that can help multi-threading in CCExtractor:

- The *APIParam* structure returned by the *api_init* function should be made thread specific data. That means, we need to modify the *api_init* function's declaration to return the thread ID of the thread it initiates and make the *APIParam* specific to the thread whose value is to be returned.
- Another way to approach the problem of avoiding dependencies on shared variable is to lock a particular variable while it is usage by another thread and thus, this variable cannot be used by another thread until it is freed by the thread that is currently using it. However, in this case, a queuing of the threads would take place and the delay could be large in cases where the present thread which has locked a particular variable is taking long to finish execution. However, this delay being large could be troublesome. Hence to avoid any delay, a better approach would be to add the *APIParam* structure returned by the *api_init* function should be made thread specific data.
- Whenever an argument is given by the user, then the user should also mention the thread id which the *api_init* function would return so that we could change only the parameters associated to a particular thread. This modification can be carried out using the thread specific commands which take input as the thread id and key of the parameter to be retrieved or modified.
- Once the parameters have been modified on the basis of arguments supplied by the user, then compilation of parameters could be done for a specific thread using its thread id.
- After the compilation has been successful, then the user can just supply the start function with the thread id the user wants to execute.
- In general, for every function in the library, another parameter specifying the thread id has to be supplied. Inside every function we can get the required parameter from the *APIParam* structure using the *pthread_getspecific* functions which take the thread id as a parameter along with a key pointing to the thread specific data item. To set a value in the thread specific data we can use *pthread_setspecific.*

Timeline (tentative)

Community Bonding period (4th May – 29th May)

- My main aim during the community bonding period would be get acquainted with the major part of the code base. As my project spans across the entire codebase, it would be really helpful to setup milestones on the basis of acquaintance with the code base and this will help a better planning of the work to be done.

- Apart from understanding the codebase, I would also like to continue solving the issues. This would help me increase my acquaintance with the code base as well as I would learn more about CCExtractor. This would definitely help me boost my interaction with the community too.

Week 1 and Week 2(30th May – 10th June)

- My work during the first two weeks mainly depends on the number of points I have gained till then. After the community bonding period, if I have had sufficient number of points to skip over solving of issues; then I would start with the development of library.
- For the development of library, the first and most important thing is to make a suitable way of compiling and linking the *api source* and *header* files to the present codebase so that the testing of the developed code for the library could be carried out easily.
- On the contrary, at the beginning of the first week, if my mentor insists on continuing solving of issues for another week, then I would take up solving issues in the first two weeks before starting with the development of libraries.

Week 3

- In this week, I would like to work on finishing the library (without multithreading) and discussing with the mentor(s) to get all the basic utilities added to the library.
- This week would just encapsulate of work related to development of CCExtractor library as a single thread.

Week 4, Week 5 and Week 6

- After the development of library in the previous week, the next important task is to extend this library to work with variable number of threads.
- Extension to multi-threading would be the most important task and it needs to be taken care by properly understanding and using the documentation of *pthreads*.
- A large portion of the development period is being allotted for this task as I believe that with proper extension to multi-threading the rest of the steps, would be easy.

Week 7

- Once the library has been developed and extended to multi-threading, then the next step would be to start with python bindings.
- In this week, I would like to work on generating the *interface file* which is needed by SWIG to generate the python bindings.

Week 8 and Week 9

- In the 8th and 9th week, I would be working on the generated python bindings using SWIG; making modifications to them and also developing new functions which would help debugging from the python interface to a greater level.

- This week would also involve testing of the multi-threading of CCExtractor in python.

Week 10 and Week 11

- After the development of the python bindings, I would like to dedicate the 10th and 11th week for documentation of the above mentioned task.
- The documentation would include a detailed documentation of the CCExtractor library that is developed in step 1 and step 2 to integrate with multi-threading.
- In addition to this, I would also like to document the python bindings and additional debugging functions for python interface.

Week 12 and week 13

- Buffer period.

Work Hours

- I have no major plans for summers and I will be able to contribute full time for 40-50 hours a week.
- My college reopens in mid-July but I will still be able to contribute full time since there won't be any exams or tests.
- My Summer break begins from 8th of May and I can start contributing therein. I would be full time available (without even any college activities) post 8th May.
- My working hours mostly would be 04:30 AM UCT to at least 11:30AM UCT.(10AM IST to 5PM IST).
- Though these working hours are temporary, I would like to work for minimum of ~45 hours a week or the time taken to complete the weekly objective; whichever being larger.

Global Result

As a global result to this project, it may be considered to **create a pip-installable Python binding of CCExtractor code.** This would greatly influence people (python developers) to make use of CCExtractor for other projects.

Evaluation Breakup

- Phase 1 Evaluation

    - During the phase 1 evaluation, a stable version of library with respect to a single thread and few features of CCExtractor library being extended to multi-threading could be considered as a point of evaluation.
    - As the phase 1 evaluation comes in between the step of *extending the library with multi-threading*, I cannot comment on complete multi-threading extension as an evaluation parameter, but some features can be considered.
    - This could be discussed with the mentor as to what all features are to be completed until phase 1 evaluation.
- Phase 2 Evaluation
    - The phase 2 evaluation could be done on the basis of following parameters:
        - A well-developed library for CCExtractor alongwith extension to multi-threading.
        - The output of SWIG interface file for basic python bindings,
        - Depending on the scenario, it may be extended to include some extra functions defined for debugging in python interface.
- Final Evaluation
    - The final evaluation could be done on the basis of following points:
        - A well-developed library for CCExtractor.
        - Extension of library to using multi-threading for launching multiple instances of CCExtractor.
        - Python bindings generation and degree of implementation.
        - Documentation of the developed library and python module.
        - A *pip* installable module for python developers and users.

How do I fit in?

- I have been working with CCExtractor since some time and I am acquainted with the basic code flow and the architecture of the database. I have tried to be an active contributor by solving issues, participating in discussions and solving queries as and when I could. I have done background study relating to my project and I am pretty much confident of getting the project finished in the span of these 13 weeks.
- Even though I haven't used all the third party supports I have mentioned in the proposal, viz. SWIG or pthreads-win32, I don't claim to be an expert with their usage experience. I have some basic knowledge about them (having tried SWIG for some sample codes too) and I would fill this gap in the pre-GSOC period and many other details could be worked out during the coding period.
- During the entire span of 13 weeks, I would be working on my laptop which has windows in it. As I am more used to Linux and vim, I have setup a vagrant environment for the development purpose. So I would be developing on the vagrant box and test the final results on windows too. Apart from these, I also own a Raspberry Pi (model 2) which could be a nice substitute in case the vagrant box fails. The pi could also be used merely for testing.

Relevant links

1. The present version of developed library (in a very basic stage) from where all the snippets have been taken could be found here from line 60 to line 495.
2. A basic guide about threading is present here. Though it is not the complete guide still it gives an overall understanding of all the details mentioned in the proposal.
3. The homepage of *pthreads-win32* which is a developed wrapper for *pthreads* which I propose to use for multi-threading.
4. The SWIG documentation could be found here. In addition, this link shows what all languages SWIG can generate the code for and which all C/C++ features are covered in it.