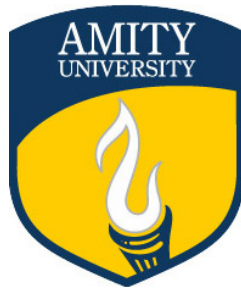


A Project Report
On
VIDEO CONVERTER SUITE USING COCOA FRAMEWORK

Submitted to



Amity University, Uttar Pradesh

In partial fulfillment of the requirements for the award of the degree of
Bachelor of Technology

In

Computer Science & Engineering

By

JAY SHARMA

Under the guidance of

DR. PRAVEEN KUMAR

DEPARTMENT OF COMPUTER SCIENCE
AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY
AMITY UNIVERSITY, UTTAR PRADESH
NOIDA (U.P.)

TABLE OF CONTENTS

I.	Introduction	8
II.	Purpose of Plan	8
III.	Background Information	9
IV.	Alternatives	10
V.	Project Goals and Objectives	11
VI.	Scope	11
VII.	Projected Budget	12
VIII.	Constraints	12
IX.	Project Management Approach	14
X.	Project Timeline	15
XI.	Project Roles and Responsibilities	16
XII.	Risk Assessment	17
XIII.	Working of the Project	18
XIV.	Project Design	29
XV.	Project Testing	31
XVI.	Results	34
XVII.	Conclusion	36
XVIII.	Future Work	37
XIX.	References	37
XX.	Appendix	38
XXI.	Synopsis	39
XXII.	Weekly Progress Reports	41

I. Introduction

The project aims to develop a lightweight macOS application to help interconvert video containers and codecs. Acceptable input would be either an uncompressed RAW y4m file or an encoded file with any of MP4/MKV/AVI containers. The output is a video file with a desired codec - libx265 or libx264 or libvpx-vp9. Application will be deployed for macOS environments 10.12 or newer. The project will achieve the conversions by interacting with the terminal and running FFmpeg command lines through shell scripts. IDE Used will be Apple Xcode 9.2 as this is the only IDE that allows macOS development. Language used will be Apple Swift 4 along with #!/bin/bash scripting.

II. Purpose of Plan

The project tries to minimize the space complexity by using cocoa framework class to run a shell script. By doing so the size of the application reduces from several gigabytes to just a few megabytes. Shell scripts are used to run commands as if they were to be running on a command prompt or terminal. The main purpose of the project is to create a lightweight application distributable that can achieve the same quality and speed of a heavy weight application while transcoding but is significantly smaller in size. The project currently gives an option of three encoders: H.265, H.264 and VP9 codecs along with three containers: MKV, MP4 and AVI.

Test outputs, thread synchronization, dispatch processes, pipelining output and NS classes are discussed in the latter half followed by graphical test analysis, constraints and a conclusion. The project is distributable along with a license and can be used for personal as well as commercial usage for all the video conversion needs that a person, group or an organization may have.

The project entitled “Video Conversion Suite using Cocoa Framework” shall be referred as “BluBolt Suite” for convenience. BluBolt Suite can also act as an effective benchmarking tool for relative comparison between encoders and containers on the basis of Peak Signal-to-Noise

Ratio (pSNR) verses Bitrate, Space Savings (SS), Space Compression Ratio (SCR) and Bitrate Savings (BS). Given any application or program, where in Space complexity is evaluated against Time complexity of the encoders, priority shall always be given to Time complexity being lower but is the possible to achieve an equilibrium between the two can be tested through BluBolt Suite.

III. Background Information

Video encoding forms the basis of videography. Given any camera, it always shoots in RAW capturing frames at a high rate and then stitching them to make it a video. In such a process, pixel information of the given camera resolution according to the sensor is captured. Every frame's every pixel is stored in binary form RGB, ranging from 8-24 bits/pixel depending on whether an alpha channel is introduced or not.

It is absolutely necessary to have a video encoding tool as the one that has been developed. Videos need to be transcoded everywhere from games rendering frames, medical imaging, videography, video chat and conferencing to entire multimedia and entertainment industry. The newer technologies with virtual reality need the faster H.265 encoders. Technology platforms like YouTube utilize Google's very own VP9 encoder for transcoding all the content uploaded on it website. Outstanding compression is required to maintain the video quality while effectively maintaining a lower bitrate. This can be achieved through newer codecs, i.e., H.265 (using libx265 library of libavutil), however where transcoding speed is required, the most effective one shall prove to be its predecessor H.264 (using libx264 library of libavutil). The older standard which is still very widely popular is H.264. It was introduced way back in 2003 but recently has not been able to keep up with increasing resolutions and bitrates. As a result, H.265 was in development and was finally introduced in 2017 as a new worldwide industry standard. It inherits most of its predecessor's features and introduces minor improvements in the way blocks are processed and as a result achieved a more compressed output while maintaining the bitrate. If the results are to be believed, H.265 is over 50% more efficient than H.264.

BluBolt Suite uses these codecs to achieve encoding. Since H.265 supports over 8K resolution (7680x4320), so does the BluBolt Suite. It effectively handles compression by multithreading and using dynamic block structure analysis. The suite is very scalable to meet today's resolution and bitrate demands in general. Since the Suite is built using Cocoa framework and runs shell scripts, there is no need to introduce heavy code and UI elements, thereby saving space on installation. The application is ~12 MB as compared to a conventional converter using ~2 GB of space. BluBolt Suite integrates FFmpeg compiled binary as a repository to run FFmpeg command line passed on as an argument to a Bash script (Type of Shell script) as sudo terminal command to achieve it. FFmpeg is an open source command line tool which can be installed and compiled using Homebrew on any UNIX system but instead a precompiled binary is utilized.

IV.Alternatives

Alternative to such a suite are available as heavy software downloads. BluBolt Suite achieves every function that a normal conversion tool can, the only difference being their extremely big size. With the dawn of SSDs, the storage though has become ~ 17 times faster but the ones with higher capacity have skyrocketing price tags. A common man is forced to buy a lower capacity SSD and installing such a heavy software could be troublesome. There are many available like Handbrake Advanced Converter, MOVAVI, AVC and many others.

Keeping this in mind, a lightweight framework, namely Cocoa framework has been used to inherit the NSClasses into the application. The application is explained in detail in the latter sections with all its functionalities and test cases performed. BluBolt Suite can also be used for performing testing of encoders for performance, Space and Time Complexity. Test results are attached in the end for reference. The project shall stay closed sourced with a distribution license.

V. Project Goals and Objectives

- A. To develop a lightweight application using Cocoa framework
- B. Integrate FFmpeg precompiled binary with Xcode 9.2 using Swift 4.0
- C. Run a Bash Shell Script for every Build
- D. Create ViewControllers and Segues for passing information between views
- E. Pipeline the output to NSScrollView
- F. Enable multithreading and process synchronization
- G. Be able to terminate transcoding in between
- H. Ability to Copy logs
- I. Create a distribution agreement
- J. Support for over 9 different configurations of codecs and containers
- K. Troubleshoot ability by directly opening Terminal from within the App
- L. Analysis on pSNR vs Bitrate for H.265, H.264 and VP9
- M. Percentage Compression achieved by encoders
- N. Find Space Compression ratios for codecs
- O. Time Complexity versus Space Complexity for output files

VI.Scope

The project assumes the user has basic knowledge of working on IDEs, video encoding and how it may be achieved. The project is coded entirely in Swift 4.0 on Xcode 9.2 along with Bash script which limit the development to macOS operating system environments only. The project can be built from scratch using this report, compiled and run for macOS 10.12 or higher which remains the deployment target.

Project outcome is a compressed transcoded video in a desired container of choice and a desired codec of choice. The options remain limited to a total of 3 each but maybe expanded to over 10 each in the next update.

VII. Projected Budget

Budget is negligible as FFmpeg is an open source binary provided for free community distribution for all major operating systems. However, after sandboxing and creating a distributable application which can be uploaded to App Store requires a developer account at US\$ 99.00 for personal edition and US\$ 299.00 for enterprise edition. Business revenue model of the project is to make the application as a paid distribution instead of relying on in app ads. The application will be priced at US\$ 0.99 per iCloud purchase or INR 75.00 per unique purchase from the Indian App Store.

VIII. Constraints

A. Development Hardware

1. Dual core 35W Intel Core i5 3210M @ 2.5 GHz (turbo boost up to 3.1 GHz)
2. 16 GB 1600 MHz DDR3L RAM
3. 256 GB Samsung 850 Pro SSD

B. Development Software and Languages

- | | |
|------------------------|----------------------|
| 1. Apple Xcode 9.2 IDE | 3. Apple Terminal |
| 2. Apple Swift 4.0 | 4. Bash Shell Script |

C. External Resources Added

- | | |
|------------------------|-------------------------------|
| 1. AppIcon.icns | 3. libswiftRemoteMirror.dylib |
| 2. BuildScript.command | 4. Main.storyboardc |

D. Frameworks Integrated

1. Cocoa.framework
2. CoreData.framework
3. Foundation.framework
4. AppKit.framework

5. Frameworks Referenced

1. libswiftAppKit
2. libswiftCore
3. libswiftCoreData
4. libswiftCoreFoundation
5. libswiftCoreGraphics
6. libswiftCoreImage
7. libswiftDarwin
8. libswiftDispatch
9. libswiftFoundation
- 10.libswiftIOKit
- 11.libswiftMetal
- 12.libswiftObjectiveC
- 13.libswiftos
- 14.libswiftQuartzCore
- 15.libswiftSwiftOnoneSupport
- 16.libswiftXPC

F. Input Constraints

1. The file to be converted must be dragged and dropped in the field or entire path location must be specified and should not have spaces
2. Output name cannot be left blank
3. By default, configuration is set to H.265, MKV but can be changed
4. Build can only be terminated during the encoding process
5. Build log can only be copied after encoding has started
6. You must agree to the license in order to copy the log
7. NSScrollView will stay locked, cannot be selected or copied but only viewed
8. Output directory location is fixed to Desktop
9. Two videos of same name cannot exist in the same output directory
10. Build shall stay locked during encoding process but may be terminated

G. Sample Test Videos

Name	Res.	Frames/Sec	T. Frames	Size
1.bridge_close_cif.y4m	CIF	30	2001	304 MB
2.ducks_take_off_444_720p50.y4m	HD	50	500	1.38 GB
3.ducks_take_off_1080p50.y4m	Full HD	50	500	1.56 GB
4.ducks_take_off_2160p50.y4m	Ultra HD	50	500	6.22 GB
5.Akiyo_CIF.y4m	CIF	30	300	45.6 MB

Table 1: Shows all the declared outlets with their classes used in the project

H. Minimum Requirements to Run the Software

1. macOS 10.12 or higher
2. 4 GB RAM RAM
3. Intel Core i5
4. HD Display

I. Other Constraints

1. For fairness, all three encoders were kept at QP 28, 32, 36 with max CU size 64 and min CU size 8
2. Only one argument will be passed at a time as a collective FFmpeg command line

IX. Project Management Approach

A phased project management model was followed during the entire development of the project. A phased model has is an enhancement of Rapid Application Development model and involves splitting up of our main task and goals to several smaller ones and be treated as module

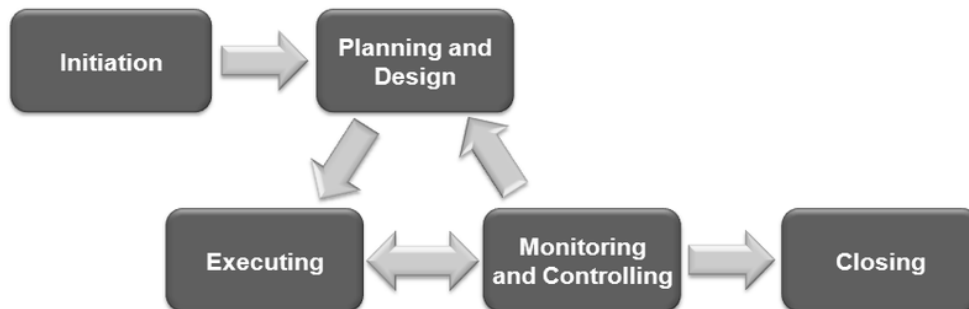


Image 1: Phased project management model (improvised RAD)

The first phases are planning, designing and actual development of the Software, named “BluBolt Suite”. Final phase involved testing of the software by encoding binary RAW video files and 45 test cases were developed for unit testing. They were monitored and results were generated and analyzed in Phase 5. All 45 test cases are presented in the latter half of the report.

X. Project Timeline

Week 01: Defined the goal and aim, performed requirement analysis for the project, decided output and deliverables, set targets to be achieved, started with research on how to implement it

Week 02: Defined a software development model to be followed for the project, prepared a rough draft for Software requirements, prepared a Pert Chart and studied about H.265/264

Week 03: Selected IDE, configured the build settings for the application in Xcode, selected a development language, started with Abstract & Intro, Interfaces that need to be overridden

Week 04: AppDelegate Class initialized, Frameworks added, built classes for ViewControllers to be linked, Main.Storyboard as a visual board for linking, Classes referenced with Storyboards

Week 05: UI elements added to Main.Storyboard, Encoder support added, Container support to be added, Documentation started, First responders disabled

Week 06: Cocoa framework - NS Views studied, Segue connections created for control passing, Modals, Popovers & Sheets implemented on Segue, @IBAction defined & @IBOutlet declared

Week 07: NSProcess() implemented, Command line tool FFmpeg integration with Xcode, Arguments created for FFmpeg via @IBOutlets, Space handling in file paths, encoding achieved

Week 08: Passed parameters as a command for FFmpeg, StdErr, StOut and StdIn defined, Dispatch, Sync and Async blocks implemented, documentation updated

Week 09: Added a Shell Script to Xcode project, Bourne Shell (!/bin/sh) vs. Bash (!/bin/bash) comparison and selection, changed Script permissions to execute, piped the output to NSScrollView, Advanced Sandboxing methodology for permission access (AppStore uploading) to be studied.

Week 10: Gave ability to stop scripting in between if the user wants to, can now copy the created log to clipboard, disabled buildButton while process is running, Add an agreement to not use code/logs for redistribution, Dismissing ViewController and passing control to the linked Controller

Week 11: Restricted access and settings - disabled views to prevent copy, TaskBar Menu to be amended to add redirected viewsw, Icon packages created in Photoshop, clean built project

Week 12: Downloaded sample y4ms for encoding, defined Testing environment for unit tests, Testing conditions and parameters were set, exported encoded files to desired output directory

Week 13: Added TabView, Added external application VLC opening command, kill processes after processing, Analyzing logs

Week 14: Extending to extract audio, Size complexity, Compression ratio, Space efficiency, Business model with cocoa

Week 15: Added conclusion, added references, added future scope and study, formatting the report and experimental result reevaluation

XI. Project Roles and Responsibilities

Project has been solely planned, designed, developed, tested, maintained by Jay Sharma, a final year student of Amity School of Engineering and Technology, Amity University, pursuing his Bachelors in Computer Science Engineering, with enrollment number A2305214054 (8CSE1), under the guidance of Dr. Praveen Kumar as part of final Major Project.

Jay Sharma is the sole owner and accountable for the success of this project. All the business test cases presented, code, workflow, statistical data, graphs, images, tables and the project as a whole are a property of Jay Sharma. All deliverables were managed, handled and submitted on time. All the planning, implementation and testing performed during the course of this project are credited to him. The project is not distributable and no part should be replicated or used without prior permission. The project is currently in testing phase. You may report bugs and problems at sharma.jay95@gmail.com

XII.Risk Assessment

Table 2: Shows risk assessment of potential problems

S. No.	Risks	Impact	Probability	Risk Rank	Impact	Solution
1	Incomplete frame in Raw y4m file	High	Rare	1	Stops encoding the file in mid	No solution, file can't be encoded
2	File with same name and container already exists in directory	Low	High	3	Encoding stops and asks to overwrite	Terminate it and restart the encoding process
3	Script not running	Low	Low	2	Incomplete output file	Encoding needs to be restarted
4	Blank Input	None	High	4	Unable to start encoding files	Escape and enter new input
5	Missing library/framework	None	Frequent	3	Unable to start encoding files	Missing libraries can be installed
6	Application not responding	Medium	Rare	2	File becomes corrupt	Terminate it and restart the application
7	File path contains spaces	None	High	4	Unable to start encoding files	Escape and enter correct path
8	Unable to locate encoded file	None	Rare	4	No impact	See package contents or search file
9	Encoding taking too long	None	Frequent	4	No impact	Change encoder
10	Unable to copy log	None	Rare	4	No impact	You can copy again
11	Cannot locate FFmpeg	High	Rare	1	Encoding cannot take place	Change path to bin if you've installed Java
12	Application could not be launched	None	Rare	4	Application doesn't start	Grant permission in system settings

XIII. Working of the Project

A. Compression definitions

1. Size Compression Ratio (S.C.R.)

$$\text{S.C.R.} = (\text{Uncompressed RAW y4m Size} / \text{Encoded Size})$$

2. Space Savings (S.S.)

$$\text{S.S.} = [1 - (\text{Encoded Size} / \text{Uncompressed RAW y4m Size})]$$

3. Bitrate Compression Ratio (B.C.R.)

$$\text{B.C.R.} = (\text{Bit-rate of RAW y4m} / \text{Bit-rate of Encoded Video})$$

4. Bitrate Savings (B.S.)

$$\text{B.S.} = [1 - (\text{Bit-rate of Encoded Video} / \text{Bit-rate of RAW y4m})]$$

B. NSTask

macOS is UNIX based operating system, with pre compiled and installed command line tools & scripting languages like Bash, Perl-V, Bourne Shell, Swift, Python and Ruby. We can also install open sourced command line tools like FFmpeg.

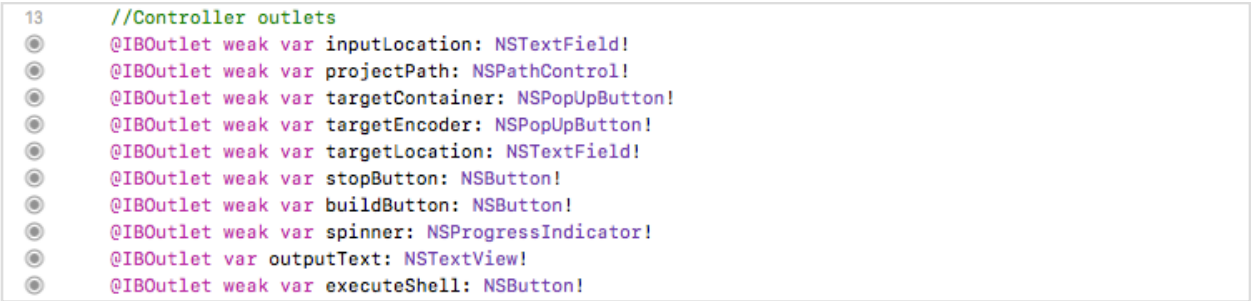
NSTask in Xcode gives a user the power to execute a different program on their system as a thread or a subprocess, as well as monitor its execution throughout while the main program is running. *NSTask* can be viewed in terms of inheritance with a parent-child relationship. A parent can create an object for child and ask it to perform functions and logically the child should obey the instructions. *NSTask* is just that and works similarly. When we instantiate a program, we give it commands & tell where to report any errors or outputs.

NSTask provides a Graphic User Interface to terminal like command line tools which may work either on the front end or as a background process. Command-line tools are extremely efficient and powerful, but the user must remember how to call it & what arguments should be passed into it every time and must type all of it. Using GUI gives control over functions without having to learn all the commands and their arguments.

C. Structure and UI

The *NSTask* will help to run FFmpeg command lines in the background. Since we're importing *Cocoa*, all the basic user interfaces are in place and our heavy job is left to do with NSTask. We have to work on our default View Controller which has been named as "*ViewController.swift*". The first step is to open *ViewController.swift* and define our outlets that we might want to reference for later.

The following @IBOutlets have been defined:



```
13 //Controller outlets
14 @IBOutlet weak var inputLocation: NSTextField!
15 @IBOutlet weak var projectPath: NSPathControl!
16 @IBOutlet weak var targetContainer: NSPopUpButton!
17 @IBOutlet weak var targetEncoder: NSPopUpButton!
18 @IBOutlet weak var targetLocation: NSTextField!
19 @IBOutlet weak var stopButton: NSButton!
20 @IBOutlet weak var buildButton: NSButton!
21 @IBOutlet weak var spinner: NSProgressIndicator!
22 @IBOutlet var outputText: NSTextView!
23 @IBOutlet weak var executeShell: NSButton!
```

Image 2: Shows all the declared outlets with their classes used in the project

All these Outlets correspond to some element in the *ViewController* which can be accessed from *Main.storyboard*. All these elements are to be declared as *@IBOutlet* or a rather simple way to do it is to open the Assistant Editor and Control-drag it in code. Both perform the same function.

A basic structure can be created by accessing the UI elements in the bottom right of Xcode for use or can also be copy-pasted so there's no need to drag drop every time. A structure has been created and the same can be accessed by opening *Main.storyboard*. The first stage is to check the functionalities we want so we can just create a framework which can later be linked to classes and files. A screenshot of the same has been shown in the image that follows this text. One point to remember is that every *ViewController* must have a separate xx.swift file where in all of its declarations and definitions may reside. These view controllers interact through each other with the help of Segue gateways which can be viewed as Modal, Popover or Sheet.

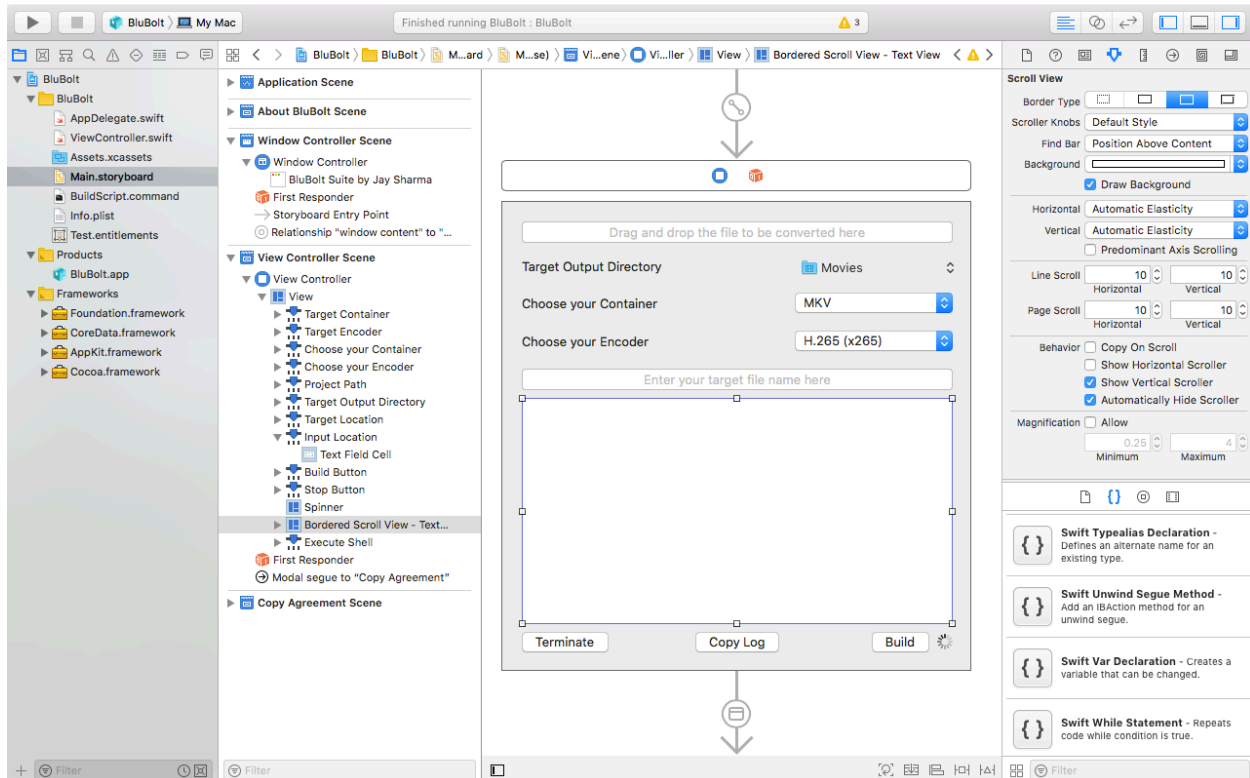


Image 3: Screenshot of Xcode 9.2 showing the basic structure of the application from Main.storyboard

D. viewDroid()

```

30  override func viewDidLoad() {
31      super.viewDidLoad()
32      stopButton.isEnabled = false
33      executeShell.isEnabled = false
34      // Do any additional setup after loading the view.
35  }

```

Image 4: Screenshot viewDroid()

viewDroid starts and sets properties to our basic elements when the application is launched. Here we want the Terminate button referenced as *stopButton* to remain disabled and be only enabled when we start building the project with the “Build” button. *executeShell* is the “Copy Log” button and should stay disabled until the project is either in the building state or has finished its build. There’s a license agreement attached to it which must be accepted in order to copy the log. The same is discussed in the sections that follow. *viewDroid* is a default function that we override from the *Cocoa* class framework.

E. *startTask()*

```
38 @IBAction func startTask(_ sender: Any) {  
39     outputText.string = ""  
40     let append1 = "/usr/local/bin/ffmpeg -i "  
41     let append2 = inputLocation.stringValue.replacingOccurrences(of: " ", with: "\\ ")  
42     var arguments:String = ""  
43     arguments.append(append1)  
44     arguments.append(append2)  
45     arguments.append(" -c:v")  
46 }
```

Image 5: Screenshot startTask()

Setting the outputText's string value to *NULL* is important as every time we need to build a new project while the application is still running, the log area must be cleared for the next usage. We have defined a new variable of type string named as "*append1*" and set the string value to the default location of our FFmpeg's command line repository from where it must be called. This is absolutely necessary to include as with incorrect path, the arguments that will be passed on to the shell script will not be able to locate it. The problem that one might face is handling spaces in project path. A space acts as delimiter and ends the entire argument in general, thus producing an error. Such a mistake can cause the entire program to crash. For handling spaces, we replace a single space occurrence with that of an escape sequence followed by a space so the space is ignored in the command line. But that raises another issue that the escape sequence escapes the string there and then, so we need to double it up to escape an escape sequence which will in turn look over the space, hence we have added "\\ " in the code.

```
47 if(targetEncoder.titleOfSelectedItem == "H.265 (x265)"){  
48     arguments.append(" libx265 ")  
49 }
```

Image 6: Screenshot of building command line

We slowly build the entire command line to be passed as an argument in our Bash shell and be run by FFmpeg by taking in the user input. Image 5 shows *targetEncoder* as a drop down list element of available encoders and we match its value using *titleOfSelectedItem* function with the value in quotes. The same process has been repeated for every encoder and every container value which will be appended later.


```

70     outputText.string = arguments
71
72     buildButton.isEnabled = false
73     stopButton.isEnabled=true
74     spinner.startAnimation(self)
75
76     var argumentsx:[String]=[]
77     argumentsx.append(arguments)
78
79     runScript(argumentsx)
80 }

```

Image 7: Continuation of startTask()

After we have appended all the variables with the help of user input, we need to display the entire command to the *outputText* as a string, so we pass all the arguments to it. Meanwhile, we also want the *buildButton* to remain disabled while encoding takes place and the *stopButton* (Terminate) to become active, so we set their visibility here. A shell script accepts arguments in the form of a [String], i.e., an array of Strings, so we need to convert our argument from String to [String] such that there's only one element in our array, which is our argument. For that, a new empty [String] array has been defined namely "*argumentsx*" in which our initial appended argument has been passed. We now finally pass on this array to our function *runScript()* which will initialize a script and run it. *runScript* function is discussed later.

F. *stopTask()*

```

83     @IBAction func stopTask(_ sender: Any) {
84         if isRunning {
85             buildTask.terminate()
86         }
87     }
25     @objc dynamic var isRunning = false
26     var outputPipe:Pipe!
27     var errorPipe:Pipe!
28     var buildTask:Process!
29

```

Image 8: stopTask()

The function is rather basic, if the process is running, it terminates the build. The function is linked to *stopButton* which is only visible while the encoding process is going on or in process. *isRunning* is as a global object with boolean value set as *false*.

G. *Building a Shell Script*

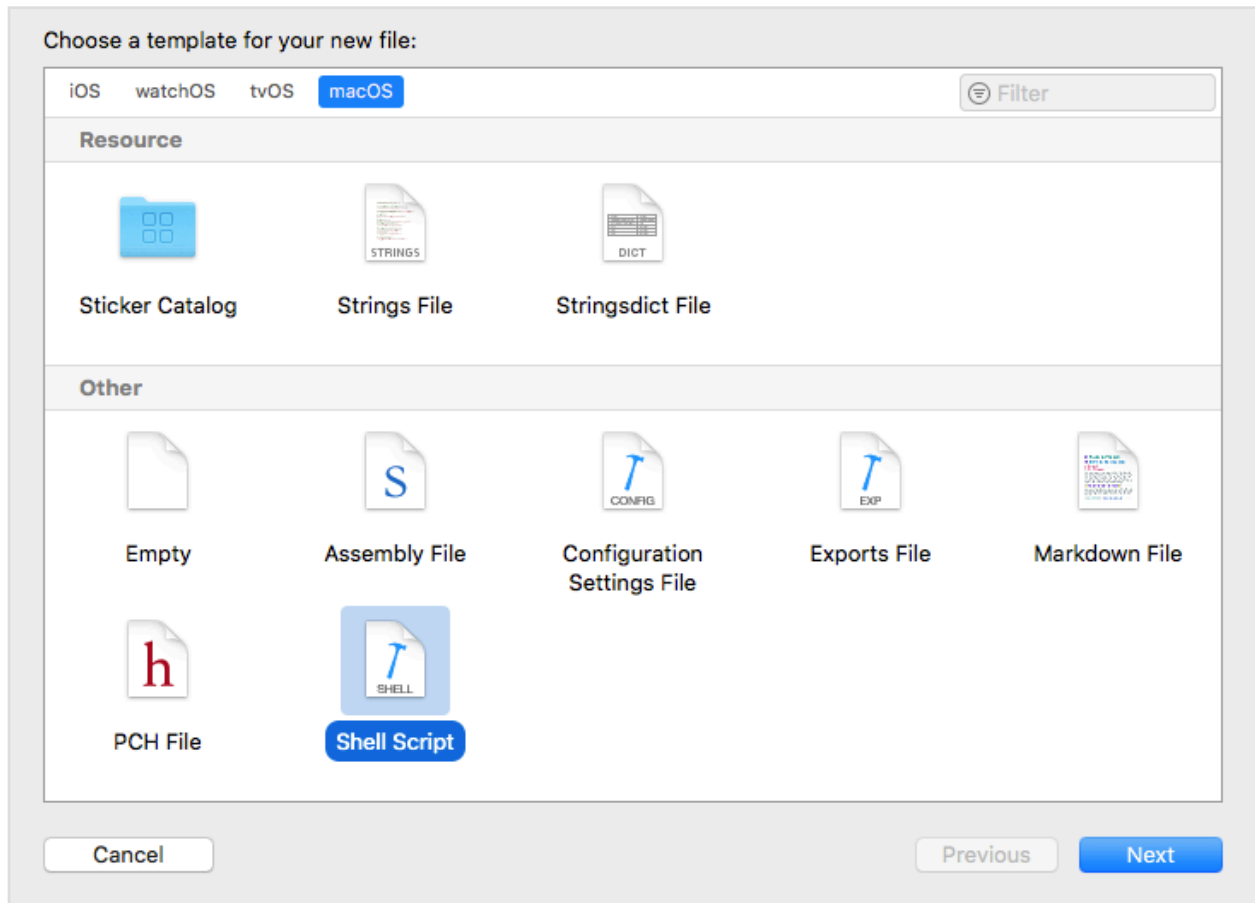


Image 9: New file templates

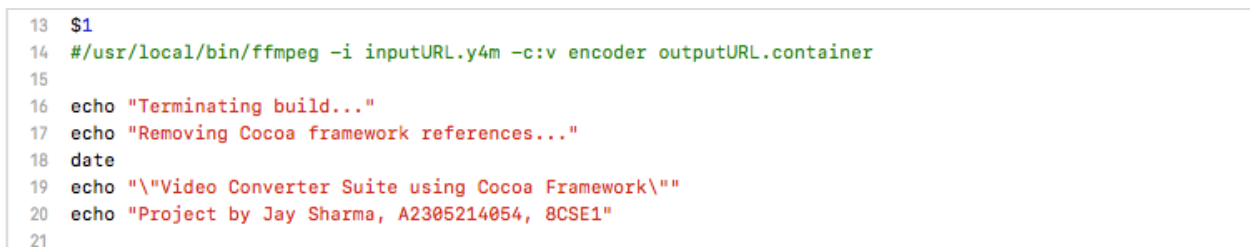
Creating a shell script is not that complicated and can be achieved by `File > New > File... > macOS > Shell Script`. It is necessary to add the script as a Resource file in our project directory so we can reference it later from within our application package. The script has been named as *BuildScript.command* and included in our Resource repository. The script however doesn't have permissions to modify anything in our system, hence the script must be given read and write permissions before we start compiling our project. A good way to do that is using "`chmod +x BuildScript.command`" in Terminal. For backwards compatibility, we have used Bash instead of a Bourne shell. Shell scripts are nothing but plain simple text files in UTF-8 that have command lines included for execution in sequential manner. Scripts are used for automation in UNIX and Linux operating system environments.



```
1 #!/bin/bash
2
3 # BuildScript.command
4 # BluBolt Converter
5 # Created by Jay Sharma
6 # Copyright © 2018 Jay Sharma. All rights reserved.
7
8 date
9 echo "Build Started"
10 echo "Beginning Build Process"
11 echo "Initating FFmpeg"
```

Image 10: BuildScript.command Shell file

The very first statement although looks like a comment but actually tells the Terminal we are using a bash shell for our script. `#` acts as a preprocessor and `!` acts as a directive to redirect the script and run from path `/bin/bash`. Code lines 3-6 are mere statements and 8-11 are UNIX/Linux commands. `date` outputs the currents date with exact time and `echo` is an output directive and prints everything written within quotes that follow it. So the script is pretty basic till now but we need to pass the argument that we created in it.



```
13 $1
14 #/usr/local/bin/ffmpeg -i inputURL.y4m -c:v encoder outputURL.container
15
16 echo "Terminating build..."
17 echo "Removing Cocoa framework references..."
18 date
19 echo "\"Video Converter Suite using Cocoa Framework\""
20 echo "Project by Jay Sharma, A2305214054, 8CSE1"
21
```

Image 11: Passing arguments in a Shell script

Arguments are passed as `$1`, `$2`, `$3` and so on starting from one. Image 11 shows line 13 accepting a single argument in Shell which will be executed as a Terminal command line. For reference, in a comment below it, sample argument that will be passed is shown. The sample code is an FFmpeg command line tool argument. It first calls FFmpeg, then takes the input file with full path, takes in the encoder, output file name with full path and container attached along with it. This argument was built with the help of user input. To finish off our shell in a brilliant manner, some echo lines are written will indicate that the encoding process is over. We need to pass on output generated to our `NSScrollView` so we can see our change-log. It can be achieved by creating a pipeline between the Script and our code through `NSPipe()`.

H. *runScript()*

```
88 func runScript(_ arguments:[String]) {
89     isRunning = true
90     let taskQueue = DispatchQueue.global(qos: DispatchQoS.QoSClass.background)
91     taskQueue.async {
92         guard let path = Bundle.main.path(forResource: "BuildScript", ofType: "command") else {
93             print("Unable to locate BuildScript.command")
94             return
95         }
96     }
```

Image 12: Running a Shell script

A new function has been defined that takes in [String] as an input which will pass on the arguments to our *BuildScript.command* shell script which accepts only [String] as arguments. When the *runScript()* function is first called, we need to change the global variable *isRunning* to true so that we can use the Terminate button along with it. This activates the Terminate button as the button is bound to the *ViewController*'s *isRunning* property through our *Cocoa* Bindings. Since we want this in our main thread, it has been put outside the queue. Then we create a *DispatchQueue.global* to run all the heavy workload as background threads in our system. Using *taskQueue.async* on *DispatchQueue* will make the application continue to process actions like clicking buttons on our main thread, scrolling on *NSScrollView* but our *NSTask* will continue to run in background as a thread till all the encoding has taken place or we have forcibly terminated the process. *Bundle.main.path* gets the path URL to our script we had named *BuildScript.command* and already had included in the application's package. For error handling an *if-else* block has been added so we can debug our program while testing.

```
97     self.buildTask = Process()
98     self.buildTask.launchPath = path
99     self.buildTask.arguments = arguments
100
101     self.buildTask.terminationHandler = {
102         task in
103         DispatchQueue.main.async(execute: {
104             self.buildButton.isEnabled = true
105             self.spinner.stopAnimation(self)
106             self.isRunning = false
107             self.executeShell.isEnabled = true
108             self.stopButton.isEnabled = false
109         })
110     })
111 }
```

Image 13: runScript continued

The next step is to create a new object of type *Process()* (Swift 4 renamed *NSProcess()* to *Process()*) and assign it to our *ViewController*'s property *buildTask*. The *launchPath* is an executable to the path we want to run our command line at. Assigning the path of our shell script to the *Process()* will help *launchPath* assign our appended argument to be passed on to *runScript*. Also, *Process()* will pass our appended arguments to the executable just like as if we had typed it into our terminal application. *Process()* also has a unique *terminationHandler* property which has a block which is run when all our tasks have finished. It also updates our UI to reflect the finished status of our *Process()*. Further we disable the buttons after the execution is complete. Notably, Build button is enabled back just like the Copy Log button but Terminate button is reverted back to disabled state. It is good to note that global *isRunning* boolean variable must also be turned back to *false* to indicate that our script has successfully run and stopped.

```
112         self.captureStandardOutputAndRouteToTextView(self.buildTask)
113         self.buildTask.launch()
114         self.buildTask.waitUntilExit()
115     }
116 }
117
```

Image 14: runScript termination

For us to run our task as well as execute our shell script, we *launch* our *Process()* object that we named *buildTask*. We can also terminate, pause and interrupt, suspend temporarily or resume *Process()*. Then we call *waitUntilExit* property which will tell the *buildTask* to not allow any further activity on our current thread till the task has been accomplished. This script and code, both are running in background as threads but the User Interface running on our main thread will not stop responding to any user input.

I. Capturing and pipelining the output

```
118 func captureStandardOutputAndRouteToTextView(_ task:Process) {
119     outputPipe = Pipe()
120     errorPipe = Pipe()
121     task.standardOutput = outputPipe
122     task.standardError = errorPipe
123
124     outputPipe.fileHandleForReading.waitForDataInBackgroundAndNotify()
125     errorPipe.fileHandleForReading.waitForDataInBackgroundAndNotify()
126 }
```

Image 15: pipelining the output

We create a *Pipe()* which attaches itself to *buildTask*'s standard input, error or the output. FFmpeg command line tool by default yields all the output as Standard Error only so to capture that, we need *standardError* but to display our script's *echo* lines, we need to capture *standardOutput*. *NSPipe* is a class that represents a normal pipe that we use in our Terminal. All that is written to *buildTask*'s *stderr* or *stdout* will be provided to this Pipe object and this is exactly what we need to capture. *Pipe()* has 2 properties: *fileHandleForWriting* & *fileHandleForReading*, both of which are *NSFileHandle* objects in *Cocoa*. We use *fileHandleForReading* to read any data that has been put in the pipe to access. *waitForDataInBackgroundAndNotify* property helps in creating and monitoring a separate background thread to check for any available data in the Pipe.

```
145     NotificationCenter.default.addObserver(forName: NSNotification.Name.NSFileHandleDataAvailable
146     errorPipe.fileHandleForReading , queue: nil) {
147         notification in
148
149         let output = self.errorPipe.fileHandleForReading.availableData
150         let outputString = String(data: output, encoding: String.Encoding.utf8) ?? ""
151
152         DispatchQueue.main.async(execute: {
153             let previousOutput = self.outputText.string
154             let nextOutput = previousOutput + "\n" + outputString
155             self.outputText.string = nextOutput
156
157             let range = NSRange(location:nextOutput.characters.count,length:0)
158             self.outputText.scrollToRangeToVisible(range)
159         })
160
161         self.errorPipe.fileHandleForReading.waitForDataInBackgroundAndNotify()
162     }
163 }
```

Image 16: displaying the output

waitForDataInBackgroundAndNotify (line 161) will notify us by calling the code we type in *NSNotificationCenter* block (line 145) to handle the output in *errorPipe* through *NSFileHandleDataAvailableNotification*. The same process must also be repeated for *outputPipe* to handle *stdout*. Within the notification handler, we get the data as *NSData* object which is then converted to a String so we can display it on our *NSScrollView*. Here, *outputText* (line 157) is a part of *NSScrollView* on our main Storyboard. On our main execution thread, we have to append the string output we got from the previous step to the tail of our text in *outputText* and so we can scroll the text area with output the

moment it arrives. This should be present on the main thread itself on not on any subthread just like all the user interactions. To get the data in the background, we will have to create a loop that should continually wait for any data present in the Pipe, process it, display it and then again wait for any available data and so on. All the data in the pipe is in UTF-8 format.

J. Copying change-log

```
166 @IBAction func executeShell(_ sender: Any) {
167     let pasteboard = NSPasteboard.general
168     pasteboard.declareTypes([NSPasteboard.PasteboardType.string], owner: nil)
169     pasteboard.setString(outputText.string, forType: NSPasteboard.PasteboardType.string)
170
171     var clipboardItems: [String] = []
172     for element in pasteboard.pasteboardItems! {
173         if let str = element.string(forType: NSPasteboard.PasteboardType(rawValue: "public.utf8-plain-text")) {
174             clipboardItems.append(str)
175         }
176     }
177     NSWorkspace.shared.launchApplication("Terminal")
178 }
```

Image 17: NSPasteboard!

executeShell is the name of the function that we have linked to the Copy Log button through our Storyboard. We will use *NSPasteboard* class to put all the contents on Copy clipboard. Here we copy contents of *outputText* of *NSScrollView* to copy all the item from it in the form of [String]. All the text is copied and pasted in UTF-8 plain text format and all the clipboard items are appended to the string in a continuous loop till all the items in our *outputText* have been read and copied. The Copy Log button is linked to another View Controller wherein a license must be accepted for the copy to be successful.

launchApplication gives the ability to any function to launch an application through *NSWorkspace*. It directly launches the application mentioned inside the quotes by matching the string from the list of applications on our system. Here we launch Terminal

XIV.Project Design

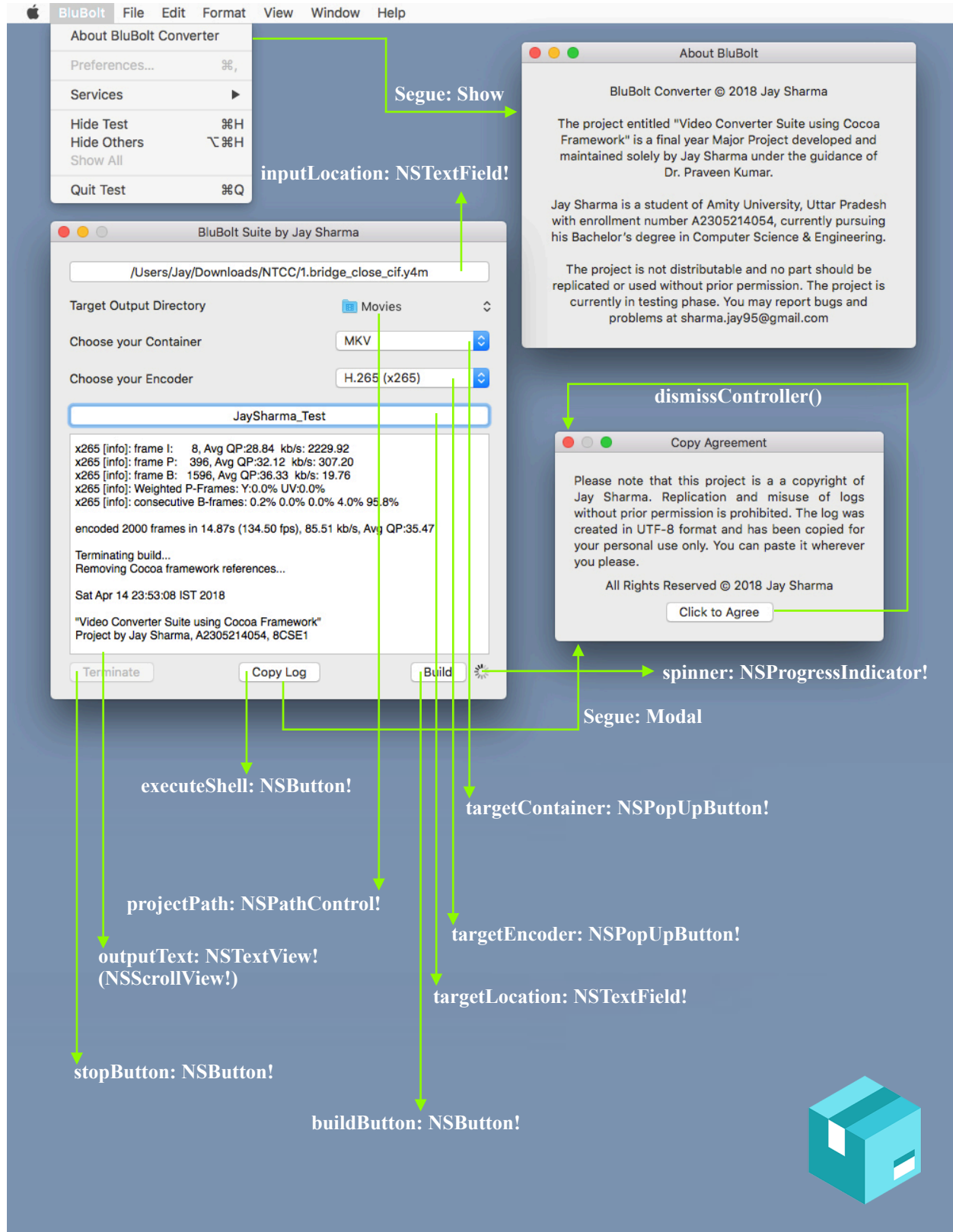


Image 18: displaying the output

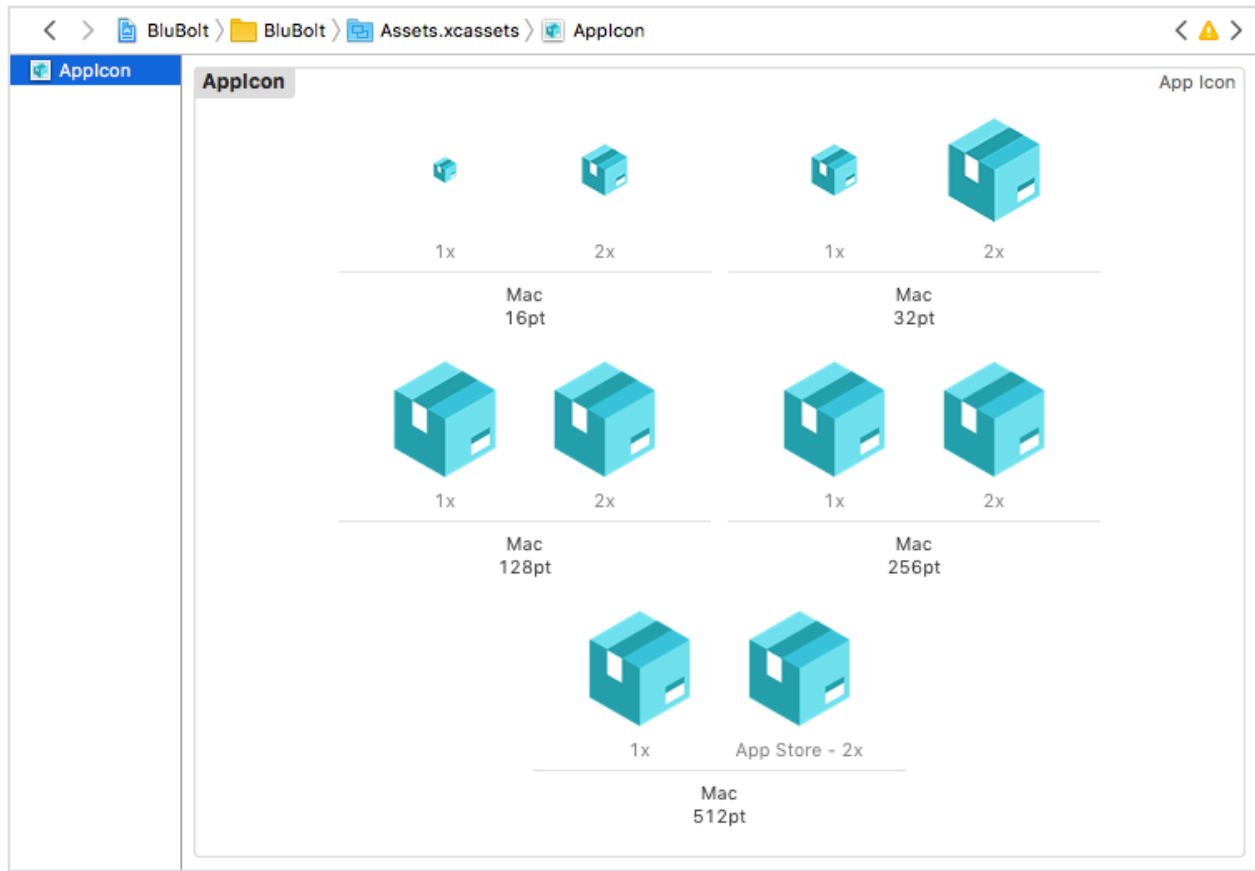


Image 19: Showing imported icon xcassets

All icon sets have been designed on Adobe Photoshop CC 2018 for macOS and imported in Xcode as icon xcassets. Project has been designed on Xcode 9.2 IDE using language Swift 4.0. All GUI and visual elements have been added through the storyboards.

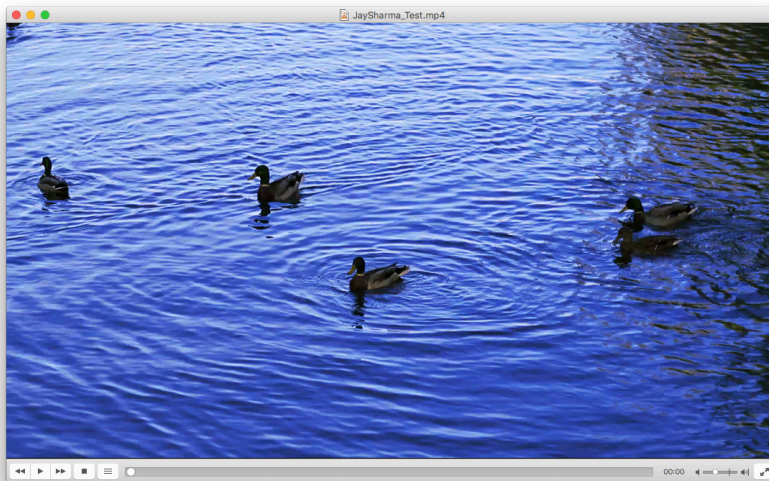


Image 20 shows a screenshot of a compressed encoded video with H.264 codec for faster processing and .mp4 as the default container.

Image 20: playing encoded file on VLC media player

TestC_006	1.bridge_close_cif.y4m	352x288	30	libx265	.mp4
TestC_007	2.ducks_take_off_444_720p50.y4m	1280x720	50	libx265	.mp4
TestC_008	3.ducks_take_off_1080p50.y4m	1920x1080	50	libx265	.mp4
TestC_009	4.ducks_take_off_2160p50.y4m	3840x2160	50	libx265	.mp4
TestC_010	5.Akiyo_CIF.y4m	352x288	30	libx265	.mp4
TestC_011	1.bridge_close_cif.y4m	352x288	30	libx265	.avi
TestC_012	2.ducks_take_off_444_720p50.y4m	1280x720	50	libx265	.avi
TestC_013	3.ducks_take_off_1080p50.y4m	1920x1080	50	libx265	.avi
TestC_014	4.ducks_take_off_2160p50.y4m	3840x2160	50	libx265	.avi
TestC_015	5.Akiyo_CIF.y4m	352x288	30	libx265	.avi
TestC_016	1.bridge_close_cif.y4m	352x288	30	libx264	.mkv
TestC_017	2.ducks_take_off_444_720p50.y4m	1280x720	50	libx264	.mkv
TestC_018	3.ducks_take_off_1080p50.y4m	1920x1080	50	libx264	.mkv
TestC_019	4.ducks_take_off_2160p50.y4m	3840x2160	50	libx264	.mkv
TestC_020	5.Akiyo_CIF.y4m	352x288	30	libx264	.mkv
TestC_021	1.bridge_close_cif.y4m	352x288	30	libx264	.mp4
TestC_022	2.ducks_take_off_444_720p50.y4m	1280x720	50	libx264	.mp4
TestC_023	3.ducks_take_off_1080p50.y4m	1920x1080	50	libx264	.mp4
TestC_024	4.ducks_take_off_2160p50.y4m	3840x2160	50	libx264	.mp4
TestC_025	5.Akiyo_CIF.y4m	352x288	30	libx264	.mp4
TestC_026	1.bridge_close_cif.y4m	352x288	30	libx264	.avi
TestC_027	2.ducks_take_off_444_720p50.y4m	1280x720	50	libx264	.avi
TestC_028	3.ducks_take_off_1080p50.y4m	1920x1080	50	libx264	.avi
TestC_029	4.ducks_take_off_2160p50.y4m	3840x2160	50	libx264	.avi
TestC_030	5.Akiyo_CIF.y4m	352x288	30	libx264	.avi
TestC_031	1.bridge_close_cif.y4m	352x288	30	libvpx-vp9	.mkv
TestC_032	2.ducks_take_off_444_720p50.y4m	1280x720	50	libvpx-vp9	.mkv
TestC_033	3.ducks_take_off_1080p50.y4m	1920x1080	50	libvpx-vp9	.mkv
TestC_034	4.ducks_take_off_2160p50.y4m	3840x2160	50	libvpx-vp9	.mkv
TestC_035	5.Akiyo_CIF.y4m	352x288	30	libvpx-vp9	.mkv
TestC_036	1.bridge_close_cif.y4m	352x288	30	libvpx-vp9	.mp4
TestC_037	2.ducks_take_off_444_720p50.y4m	1280x720	50	libvpx-vp9	.mp4
TestC_038	3.ducks_take_off_1080p50.y4m	1920x1080	50	libvpx-vp9	.mp4
TestC_039	4.ducks_take_off_2160p50.y4m	3840x2160	50	libvpx-vp9	.mp4
TestC_040	5.Akiyo_CIF.y4m	352x288	30	libvpx-vp9	.mp4
TestC_041	1.bridge_close_cif.y4m	352x288	30	libvpx-vp9	.avi
TestC_042	2.ducks_take_off_444_720p50.y4m	1280x720	50	libvpx-vp9	.avi
TestC_043	3.ducks_take_off_1080p50.y4m	1920x1080	50	libvpx-vp9	.avi
TestC_044	4.ducks_take_off_2160p50.y4m	3840x2160	50	libvpx-vp9	.avi
TestC_045	5.Akiyo_CIF.y4m	352x288	30	libvpx-vp9	.avi

Table 4: Corresponding Size Compression Ratio and Space Savings for Input Test Cases

Video No.	Cont.	Encoder	RAW Size (bytes)	Encoded Size (bytes)	SCR	SS %
1	MKV	H.265/HEVC	30,41,40,044	7,37,303	412.503	99.758
	MP4		30,41,40,044	7,45,059	408.209	99.755
	AVI		30,41,40,044	7,89,580	385.192	99.740
2	MKV	H.265/HEVC	1,38,24,03,040	67,97,675	203.364	99.508
	MP4		1,38,24,03,040	68,00,149	203.290	99.508
	AVI		1,38,24,03,040	68,12,810	202.912	99.507
3	MKV	H.265/HEVC	1,55,52,03,036	1,44,08,757	107.935	99.074
	MP4		1,55,52,03,036	1,44,11,179	107.916	99.073
	AVI		1,55,52,03,036	1,44,23,758	107.822	99.073
4	MKV	H.265/HEVC	6,22,08,03,036	4,13,26,110	150.530	99.336
	MP4		6,22,08,03,036	4,13,28,454	150.521	99.336
	AVI		6,22,08,03,036	4,13,40,856	150.476	99.335
5	MKV	H.265/HEVC	4,56,21,044	63,123	722.733	99.862
	MP4		4,56,21,044	64,647	705.695	99.858
	AVI		4,56,21,044	75,418	604.909	99.835
6	MKV	H.264	30,41,40,044	35,81,603	84.917	98.822
	MP4		30,41,40,044	35,91,324	84.687	98.819
	AVI		30,41,40,044	36,21,540	83.981	98.809
7	MKV	H.264	1,38,24,03,040	2,14,31,822	64.502	98.450
	MP4		1,38,24,03,040	2,14,34,277	64.495	98.449
	AVI		1,38,24,03,040	2,14,45,218	64.462	98.449
8	MKV	H.264	1,55,52,03,036	4,35,86,515	35.681	97.197
	MP4		1,55,52,03,036	4,35,88,901	35.679	97.197
	AVI		1,55,52,03,036	4,35,99,680	35.670	97.197
9	MKV	H.264	6,22,08,03,036	16,27,34,532	38.227	97.384
	MP4		6,22,08,03,036	16,27,35,826	38.226	97.384
	AVI		6,22,08,03,036	16,27,47,170	38.224	97.384
10	MKV	H.264	4,56,21,044	1,38,414	329.598	99.697
	MP4		4,56,21,044	1,40,072	325.697	99.693
	AVI		4,56,21,044	1,48,800	306.593	99.674
11	MKV	VP9	30,41,40,044	18,89,839	160.934	99.379
	MP4		30,41,40,044	18,83,390	161.485	99.381
	AVI		30,41,40,044	19,29,276	157.645	99.366
12	MKV	VP9	1,38,24,03,040	1,41,14,949	97.939	98.979
	MP4		1,38,24,03,040	1,41,17,693	97.920	98.979
	AVI		1,38,24,03,040	1,41,29,004	97.842	98.978

13	MKV	VP9	1,55,52,03,036	2,89,97,983	53.631	98.135
	MP4		1,55,52,03,036	2,90,00,331	53.627	98.135
	AVI		1,55,52,03,036	2,90,11,710	53.606	98.135
14	MKV	VP9	6,22,08,03,036	10,20,30,421	60.970	98.360
	MP4		6,22,08,03,036	10,20,32,340	60.969	98.360
	AVI		6,22,08,03,036	10,20,41,713	60.963	98.360
15	MKV	VP9	4,56,21,044	1,00,229	455.168	99.780
	MP4		4,56,21,044	1,02,100	446.827	99.776
	AVI		4,56,21,044	1,12,329	406.138	99.754

Table 5: Shows encoding time in seconds taken by various codecs during encoding by BluBolt Suite

Video No.	x264 (secs)	x265 (secs)	libvpx-vp9 (secs)	RAW Size	Container
1	11.32	14.17	466.56	304 MB	MKV
	11.79	14.38	532.70	304 MB	MP4
	12.25	13.96	556.21	304 MB	AVI
2	75.38	89.43	3289.48	1.38 GB	MKV
	77.26	89.66	3368.13	1.38 GB	MP4
	80.92	101.35	3542.44	1.38 GB	AVI
3	155.81	166.99	6720.84	1.56 GB	MKV
	160.45	172.84	6932.21	1.56 GB	MP4
	163.84	188.24	7524.69	1.56 GB	AVI
4	670.80	743.90	34227.62	6.22 GB	MKV
	674.63	752.90	37439.26	6.22 GB	MP4
	680.29	767.98	40103.27	6.22 GB	AVI
5	1.88	1.92	56.2	45.6 MB	MKV
	1.92	2.14	65.64	45.6 MB	MP4
	2.04	2.32	71.21	45.6 MB	AVI

XVI.Results

While analyzing the results from the 45 input test cases, it be fairly said that the Video Converter Suite can also act as a benchmarking tool apart from the mainstream converter in general. Here three different video encoding standards have been compared on the parameters of encoding time, size compression ratio and space savings with respect to gradually increasing resolution. Graphical analysis of the same is as follows:

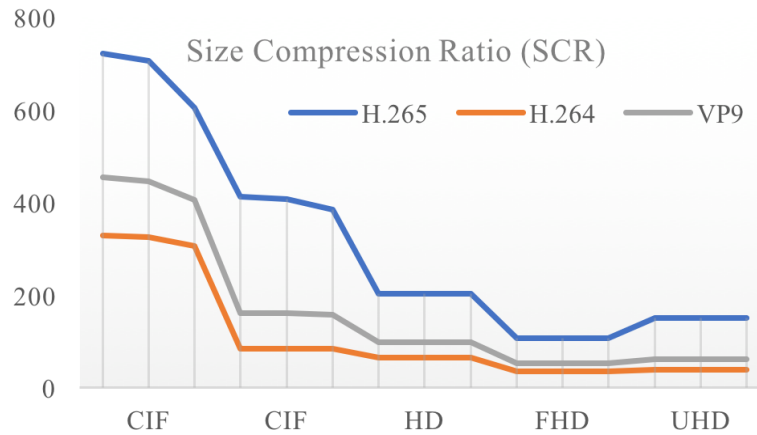


Image 22: Shows SCR compiled from 45 Input test cases; (Higher is better) y-axis: SCR

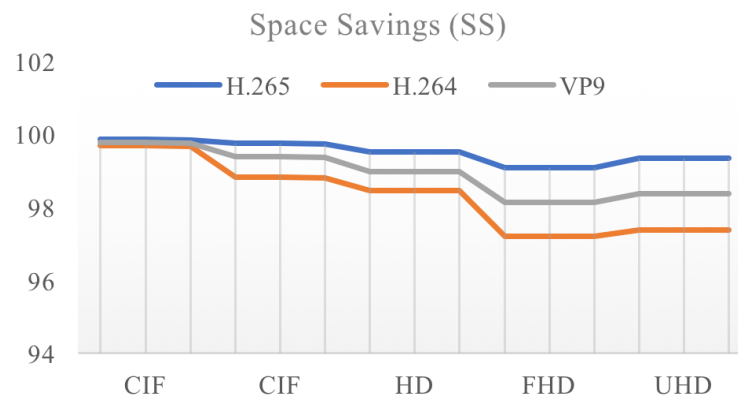


Image 23: Shows SS compiled from 45 Input test cases; (Higher is better) y-axis: SS%

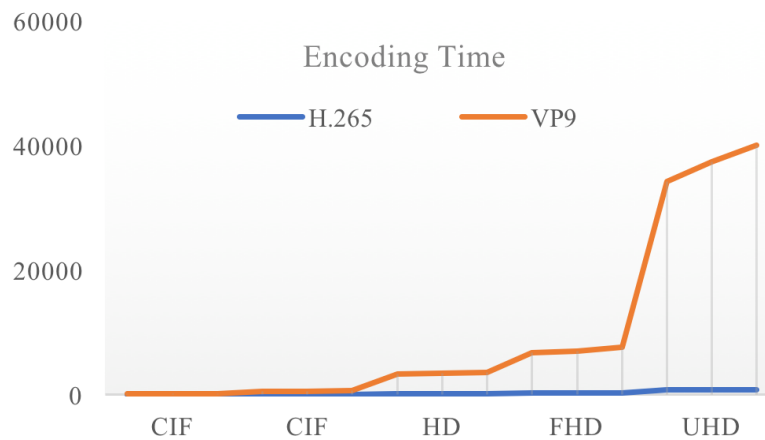


Image 24: Shows encoder VP9 taking roughly 35-45 times more encoding time with similar configuration (Lower is better) y-axis: time in secs

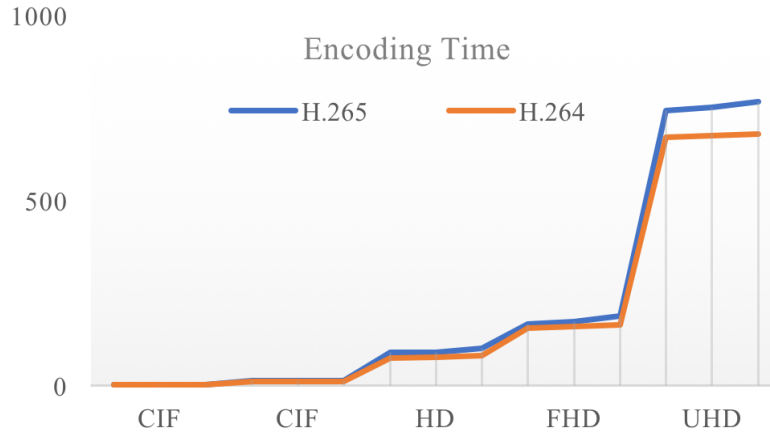


Image 25: Shows H.264 taking 15-25% less encoding time in comparison to the newer H.265 codec with similar configuration; (Lower is better) y-axis: time in secs

Time Complexity is always given a preference over Space Complexity. For testing, encoding was performed using 4 threads on a dual core processor Intel 3210M. VP9 is a codec developed by Google for its YouTube platform. The codec resides on cloud and is implemented by server workstations with extremely powerful processors. Such codecs do not perform well on local systems and take up more 35-45 times the encoding time in comparison to others. The newer H.265 codec produces a better quality video at same bitrate but takes up around 10-15% more encoding time. Dynamic block structure of H.265 proves to be more space efficient as well.

XVII.Conclusion

Video Converter Suite using Cocoa Framework is an extremely lightweight software (~12 MB) which achieves inter-conversion of encoding standards and containers as well as helps converting RAW binary frames to a compressed encoded video format. The suite can also be used as a benchmarking tool as shown in results. Since the suite has been created using open source libraries, it can be easily redistributed after sandboxing and is ready to be uploaded on AppStore. It helps a user to achieve faster conversions without the hassle of remembering command lines and typing them into Terminal every time. This has been achieved by running a bash shell script in background to automate the job. Application is optimized for multi-threading.

XVIII.Future Work

1. Add support for more containers
2. Add support for more encoders
3. Add Audio ripping tabbed view, video splitting tabbed view and video joining tabbed view
4. Create a TouchBar class for added functionality in latest MBP lineups
5. Sandboxing
6. Integrate with iCloud and Apple Pay
7. Upload on AppStore
8. Business model for distribution
9. Add directory path locator

XIX.References

1. Jay Sharma, Dr. Tanupriya Choudhury, “Study on H.265/HEVC against VP9 and H.264 on Space and Time complexity for codecs”, “IC3IoT 2018 - International Conference on Communication, Computing & Internet of Things” with ID: 149, IEEE Conference Record Number: 42386 and Electronic ISBN: 978-1-5386-2459-3
2. Md Abu Layek, Ngo Quang Thai, Md Alamgir Hossain, “Performance analysis of H.264, H.265, VP9 and AV1 video encode”, 19th APNOMS, pp. 322-325, September 2017
3. A. Dhanalakshmi, G. Nagarajan, “Enhanced hybrid quality prediction model for 8K UHD H.265 video using ANFIS”, 4th ICACCS, pp. 1-6, January 2017
4. Jianing Li, Zhaohui Li, Dongmei Li, “Performance comparison of H.264 and H.265 encoders for 4K video sequences”, 2nd IEEE Int. Conference on Computer and Communications (ICCC), pp. 531-536, October 2016
5. Juraj Bienik, Miroslav Uhrina, “Performance of H.264, H.265, VP8 and VP9 Compression Standards for High Resolutions”, 19th International Conference on Network-Based Information Systems, December 2016
6. NSTask()
<https://www.raywenderlich.com/125071/nstask-tutorial-os-x>
7. Mastering Swift 4.0
<https://www.safaribooksonline.com/library/view/mastering-swift-4/9781788477802/626e77de-4bec-4098-bbf4-b7abc080e3b7.xhtml>

8. macOS Controls

<https://www.raywenderlich.com/149295/macos-controls-tutorial-part-12>

9. FFmpeg Command lines

<https://www.labnol.org/internet/useful-ffmpeg-commands/28490/>

10. OSX Frameworks

https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/OSX_Technology_Overview/SystemFrameworks/SystemFrameworks.html

11. Cocoa Application Layer

https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CocoaApplicationLayer/CocoaApplicationLayer.html

12. Development Guide

<https://developer.apple.com/library/content/navigation/>

13. Swift Documentation

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309

14. Libavcodec Documentation:

<https://www.ffmpeg.org/libavcodec.html>

15. VP9 Documentation

<https://www.webmproject.org/vp9/>

16. H.265

<http://x265.org/hevc-h265/>

XX. Appendix

S. No.	Name	Description	Page No.
1	Table 1	Shows all the declared outlets with their classes used in the project	13
2	Table 2	Shows risk assessment of potential problems	17
3	Table 3	Input Test cases	31
4	Table 4	Corresponding Size Compression Ratio and Space Savings for Input Test Cases	33
5	Table 5	Shows encoding time in seconds taken by various codecs during encoding by BluBolt Suite	34
6	Image 1	Phased project management model (improvised RAD)	14
7	Image 2	Shows all the declared outlets with their classes used in the project	19

8	Image 3	Screenshot of Xcode 9.2 showing the basic structure of the application from Main.storyboard	20
9	Image 4	Screenshot viewDidLoad()	20
10	Image 5	Screenshot startTask()	21
11	Image 6	Screenshot of building command line	21
12	Image 7	Continuation of startTask()	22
13	Image 8	stopTask()	22
14	Image 9	New file templates	23
15	Image 10	BuildScript.command Shell file	24
16	Image 11	Passing arguments in a Shell script	24
17	Image 12	Running a Shell script	25
18	Image 13	runScript continued	25
19	Image 14	runScript termination	26
20	Image 15	pipelining the output	26
21	Image 16	displaying the output	27
22	Image 17	NSPasteboard!	28
23	Image 18	displaying the output	29
24	Image 19	Showing imported icon xcassets	30
25	Image 20	Playing encoded file on VLC media player	30
26	Image 21	UML diagram for Cocoa framework component of NSView	21
27	Image 22	Shows SCR compiled from 45 Input test cases; (Higher is better) y-axis: SCR	35
28	Image 23	Shows SS compiled from 45 Input test cases; (Higher is better) y-axis: SS%	35
29	Image 24	Shows encoder VP9 taking roughly 35-45 times more encoding time with similar configuration (Lower is better) y-axis: time in secs	35
30	Image 25	Shows H.264 taking 15-25% less encoding time in comparison to the newer H.265 codec with similar configuration; (Lower is better) y-axis: time in secs	36