Semester Long Internship Report

On

Scilab Octave Toolbox

Submitted by

Karan
GGSIPU, Delhi

Under the guidance of

Prof. Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

Prof. Kumar Apaiah
Chemical Engineering Department
IIT Bombay

Mentored By

Mr. Rupak Rokade
IIT Bombay

July 10, 2021

Acknowledgment

The internship opportunity we had with the FOSSEE Team, IIT BOMBAY, was a great chance for learning and professional development. Therefore, we consider ourselves as very lucky individuals as we were provided with an opportunity to be a part of it. We are also grateful for having a chance to meet so many wonderful people and professionals across the country who led us through this internship period.

We are using this opportunity to express our deepest gratitude and special thanks to Prof. Kannan M. Moudgalya, head of FOSSEE team, IIT Bombay, for giving us an opportunity to be a part of this project. We express our deepest thanks to Prof. Kumar Appaiah, professor in the Department of Electrical Engineering, IIT Bombay, for taking part in useful decisions and giving necessary advices and guidance to make life easier. We choose this moment to acknowledge his contribution gratefully.

It is our radiant sentiment to place on record our best regards, deepest sense of gratitude to our mentor, Mr. Rupak Rokade for the continuous support which was extremely valuable for our study both theoretically and practically and helping us to learn a lot many things.

We perceive this opportunity as a big milestone in our career development. We will strive to use gained skills and knowledge in the best possible way, and we will continue to work on their improvement, in order to attain desired career objectives. Hope to continue cooperation with all of you in the future.

Contents

1 Introduction **4**
- 1.1 About Scilab Octave Toolbox . . . 4
- 1.2 Version Control . . . 5
- 1.3 Travis CI: Continuous Integration . . . 5
- 1.4 Doxygen . . . 6

2 Scilab Octave Toolbox Structure **7**
- 2.1 File Structure . . . 7
  - 2.1.1 builder.sce . . . 7
  - 2.1.2 loader.sce . . . 7
  - 2.1.3 demos . . . 8
  - 2.1.4 etc . . . 8
  - 2.1.5 jar . . . 8
  - 2.1.6 cleaner.sce . . . 8
  - 2.1.7 unloader.sce . . . 8
  - 2.1.8 .travis.yml . . . 9
  - 2.1.9 Doxyfile . . . 9
  - 2.1.10 help . . . 9
  - 2.1.11 macros . . . 9
  - 2.1.12 sci_gateway . . . 9
  - 2.1.13 src . . . 9
  - 2.1.14 tests . . . 10
  - 2.1.15 thirdparty . . . 10
- 2.2 Guidelines to use the toolbox . . . 10

3 Contributions **12**
- 3.1 Task 1: Error Handling . . . 12
  - 3.1.1 Problem Statement . . . 12
  - 3.1.2 Role & Contributions . . . 12
  - 3.1.3 Conclusion . . . 14
- 3.2 Task 2: Structure Support . . . 15
  - 3.2.1 Problem Statement . . . 15
  - 3.2.2 Role & Contributions . . . 15
  - 3.2.3 Conclusion . . . 18
- 3.3 Task 3: Windows OS Support . . . 19
  - 3.3.1 Problem Statement . . . 19

3.3.2 Role & Contributions 19
3.3.3 Conclusion 20

4 Conclusion **21**

5 Reference **22**

Chapter 1

Introduction

SCILAB is an open-source numerical, programming, and graphics environment available for free from the French Government's "Institut Nationale de Recherche en Informatique et en Automatique - INRIA (National Institute for Informatics and Automation Research)." It is similar in operation to MATLAB and other existing numerical/graphic environments and can be run using a variety of operating systems including UNIX, Windows, Linux, etc. It includes a large number of intrinsic numeric, programming, and graphics functions.

GNU Octave is a high-level, open-source language, primarily intended for numerical computations. It provides a convenient command-line interface for solving linear and nonlinear problems numerically, and for performing other numerical experiments. It is easily extensible and customizable via user-defined functions written in Octave's language, or using dynamically loaded modules written in C++, C, Scilab, or other languages.

1.1 About Scilab Octave Toolbox

FOSSEE Octave toolbox is a toolbox in scilab maintained and developed by FOSSEE (Free and Open Source Software in Education), IIT Bombay. It is basically a toolbox which aims to bring the power of the octave right inside the scilab. So, we can call all the octave functions directly from scilab using this.

The Octave toolbox for Scilab makes use of both the Octave-C++ API and the Scilab-C++ API. Firstly, the input is fetched using the Scilab API and stored (Initially, only int/double/complex data type was supported), the input is then sent to the fun() function where the input data is passed to Octave's API for computation. The returned data is then checked to see for a successful response or an error. If an error is encountered, a general error is thrown in Scilab (Initial case), or if the computation is successful, then the output is sent back using Scilab's API. It can solve the following types of problems:

- Linear algebra problems
- Finding the roots of nonlinear equations
- Integrating ordinary functions

- Manipulating polynomials
- Integrating ordinary differential and differential-algebraic equations etc.

FOSSEE Octave toolbox mainly uses 2 octave libraries, namely :

- loctinterp
- loctave

1.2 Version Control

Git is a version control system for tracking changes happening in files of computers which comes along with web-based hosting services for repositories. It can be used to coordinate work among multiple people across the world. We have made the toolbox in our systems as git repositories (local repository) to push it to GitHub (to create a remote repository).

GitHub is a web-based hosting service for version control using Git which offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. We have used GitHub for version control using Git. We used Github for two more major purposes that are:

- Linking it with Travis CI, the continuous integration tool, where we can deploy and test our projects which are in Github.
- Adding Doxygen, which is the de-facto standard tool for generating documentation from annotated C++ sources.

The file ".travis.yml" in Github indicates the Travis CI to test our projects whereas the "Doxyfile" indicates the file to configure Doxygen documentation.

1.3 Travis CI: Continuous Integration

Travis CI is a continuous integration tool, where we can deploy and test our projects which are in Github. We have linked our repository to our Travis CI account and then we initiated a build in Travis. Travis just looks into the .travis.yml file in the repository, and based on the script that we have mentioned, it builds our project.

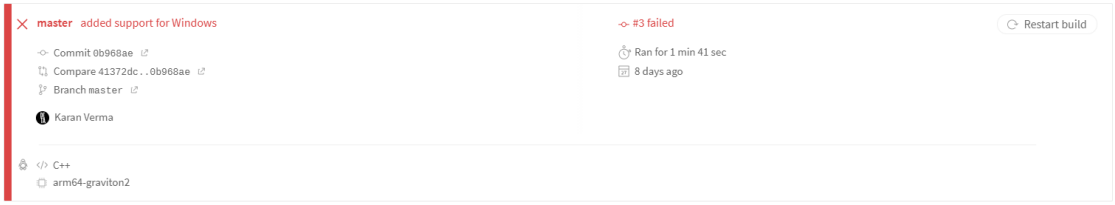


Figure 1.1: Travis Build Log

We have built the test.sce of the toolbox in Travis CI, which is used to validate the functions present in the toolbox. On successful completion of the build we get

an indication of build success whereas if the build fails, then we get an indication of build failure which indicates that we have to revisit the test file to fix the errors or bugs. After every build, be it successful or failed builds, we get notification mails from Travis regarding the commit information and the build status. So each time, we triggered the builds by entering the desired commit messages of the git repository, to test the toolbox.

1.4 Doxygen

Doxygen is a popular tool to document the code which can be used to generate code for a variety of languages. It is great at generating the documentation for the class definitions (the member variable, methods, etc.), class hierarchies (inheritance hierarchy), etc. But, it does not do much with documenting the algorithm (which is typically there in .cpp files). There are two main steps in using Doxygen:

- To use Doxygen, we write comments in code using the format that Doxygen understands. The comments are included in the header files (.h) files. But, we should also comment code in your .cpp files, though Doxygen won't use them extensively.
- Then, simply run Doxygen, which generates an html folder (in our case, inside the doc folder in Github) with the index.html file in it. The documentation for the code is now in an easy to read html file.

Chapter 2

Scilab Octave Toolbox Structure

Followings are the list of visible files and folders in the main directory of the fossee-scilab-octave-toolbox of Scilab:

- builder.sce
- loader.sce
- demos
- etc
- jar
- cleaner.sce
- unloader.sce
- README.md
- .travis.yml
- Doxyfile
- help
- doc
- locales
- macros
- sci_gateway
- src
- tests
- thirdparty

2.1 File Structure

2.1.1 builder.sce

This file builds the macros, help and the loader.sce files. Type the command exec builder.sce in the scilab console to execute this file. Both these have to be executed every time, to load the toolbox for usage.

2.1.2 loader.sce

It is basically a file that calls the function scilab_octave.start present in the etc folder. This has to be executed first on opening the toolbox, using the command exec loader.sce in the scilab console.

2.1.3 demos

It contains demo files for the toolbox.

2.1.4 etc

etc directory contains the initialization and finalization script of the toolbox which runs at the beginning and termination of the toolbox. They are executed while executing the loader and unloader files.

scilab_octave.start The name of the initialization script for a toolbox is the name of the toolbox followed by ".start". It is executed when we run the loader.sce file. It's purpose includes :

- Load function libraries from macros directory.
- Load gateway and shared libraries form sci_gateway and thirdparty directory.
- Load help from help directory.
- Load demos from demos directory.

scilab_octave.quit The name of the finalization script for a toolbox is the name of the toolbox followed by ".quit". It is executed when we run the loader. It's purpose includes :

- Unlink the toolbox libraries.
- Remove any preferences that were set by the toolbox.

2.1.5 jar

This folder has a scilab_en_US.jar file. This file is basically a Java Archive package file format typically used to aggregate many Java class files and associated metadata and resources (text, images etc.) into a file for distribution.

2.1.6 cleaner.sce

This file is generated by builder.sce. On executing this file, we actually delete the loader.sce and the unloader.sce files. One caution to the users is that do not edit this file.

2.1.7 unloader.sce

It is used to unload the toolbox.

2.1.8 .travis.yml

This file is included to facilitate the builds we can trigger in Travis CI. Travis CI provides a default build environment and a default set of steps for each programming language. We can customize any step in this process in .travis.yml. .travis.yml can be very minimalistic or have a lot of customization in it. So each time when a build is triggered by the user, an new scilab instance is opened in the terminal without any gui (scilab-cli) and executes test.sce in the tests folder of the toolbox (because of the -f flag used here).

2.1.9 Doxyfile

Doxygen uses a configuration file to determine all of its setting. This configuration file is a free-form ASCII text file with a structure that is similar to that of a Makefile, with the default name Doxyfile. It is parsed by Doxygen and essentially, consists of a list of assignment statements. Each statement consists of a TAG_NAME written in capitals, followed by the equal sign (=) and one or more values.

2.1.10 help

The help section that covers all the functions that the toolbox currently consists of.

2.1.11 macros

Macros folder contains scilab function files. Files with extensions other than sci will not be compiled when the builder is run. Scilab macros can be:

- A Scilab function file which returns the result after computation.
- A Scilab function which calls a C, C++ or FORTRAN code.
- A Scilab function which calls a binary library.

2.1.12 sci_gateway

Now the sci_gateway directory contain all necessary files to create the builder for the primitive octave_fun. For this, the builder file is in the sci_gateway/cpp/ directory, this builder (named builder_gateway_cpp.sce) creates the new shared libraries to link the compiled C and new Scilab interface routines and generates a loader. This loader file calls the addinter function to load dynamically the shared library.

2.1.13 src

The src directory contain all source code files for the octave_fun function i.e. to call octave from c and then pass the result to scilab.

2.1.14 tests

This folder contains 2 tests file that are used to validate all the functions present in the toolbox macros. One is the demo.sce file which is used for the demo purposes whereas other is the test.sce file which is executed in the Travis CI (Continuous Integration).

2.1.15 thirdparty

It contains the header files and dynamic link libraries for the fun function for both, Linux and Windows.

2.2 Guidelines to use the toolbox

The toolbox is available at Github. The users are requested to go through the README.md file prior to using the toolbox.

Else, the users are requested to follow the following steps:

- **Prerequisites for Linux**

  1. `sudo apt-get install build-essential` (117 MB download)
  2. `sudo apt-get install liboctave-dev` (103 MB download)
  3. `sudo apt-get install octave`
  4. `sudo apt-get install scilab`
  5. Now, install the required octave packages using the below command in linux terminal `sudo apt-get install octave-pkg name`
     For example, to install signal package in octave, do
     `sudo apt-get install octave-signal`

- **Prerequisites for Windows**

  1. Download and Install Scilab 6.0.1 x64 from Scilab.org
  2. Download and Install Octave 4.4.1 x64.
  3. Install Mingw Tollbox for Scilab (`https://atoms.scilab.org/toolboxes/mingw/0.10.5`).
  4. Create an user variable called `OCTAVE_HOME` with value equal to the installation directory of Octave, (Default: `C:\Octave\Octave-4.4.1\`).

Now build and load the toolbox using the folllowing steps:

1. Clone the repository as it is.
2. Go to the root folder and execute the command `exec builder.sce` to build the toolbox.

3. Load the toolbox using `exec loader.sce` and start using the functions in the toolbox.

This step should be repeated every time you restart the Scilab to load the toolbox again. Once the toolbox is built and loaded by following the steps mentioned above, we can verify the functioning of the toolbox by executing the test.sce by `exec tests/test.sce`.

Chapter 3

Contributions

3.1 Task 1: Error Handling

3.1.1 Problem Statement

Capturing the original octave error messages and returning them to Scilab.

This toolbox basically makes use of both the Octave-C++ API and the Scilab-C++ API. So, firstly, the input is fetched using the Scilab C++ API, which is then sent to the fun() function where the data is passed to Octave's API for computation using the feval function. The returned data is then checked to see for a successful response or an error. Now earlier, if an error was encountered in the octave side of this toolbox, it threw only following errors, instead of the specific error by the octave:

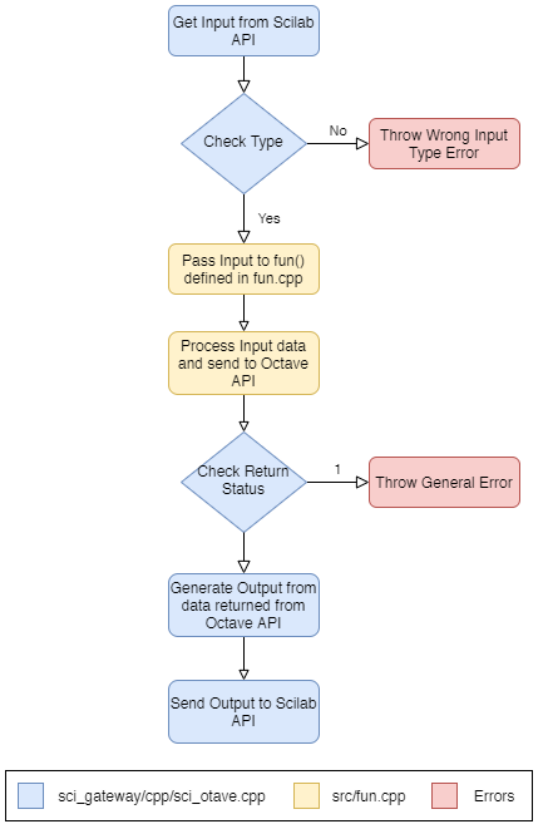
- Octave interpreter exited with status = ...
- Error encountered in Octave evaluator!
- Octave unable to process!

3.1.2 Role & Contributions

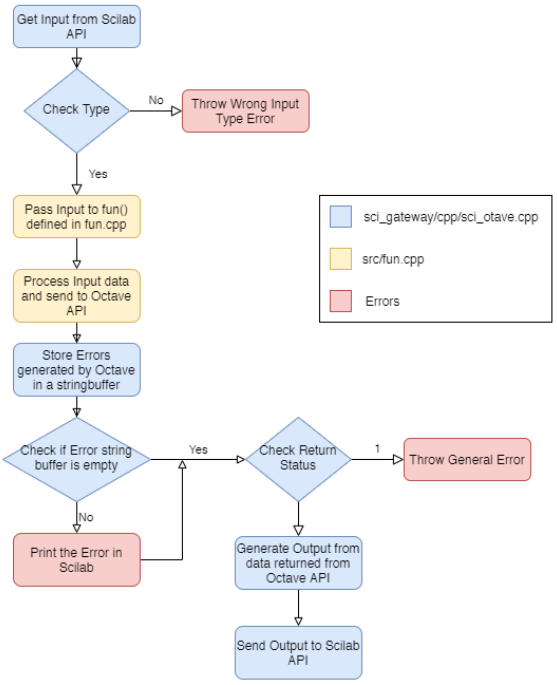
Role: Documentor

For the major part I pertained to creating documentation for the toolbox. I went about creating documentation with the following approaches:

- Internal Toolbox Document
- Doxygen: For auto generated documentation from Source Code



(a) Original Working of the Toolbox



(b) Working of the Toolbox with Error Handling

Figure 3.1: Toolbox Flowchart

Figure 3.2: HTML Homepage of Doxygen Generated Documentation

3.1.3 Conclusion

In the current implementation of error handling in Octave's interpreter, the errors were being sent into an error buffer by Octave. The developer of this task came with a neat approach to store the errors from the buffer and send the same to Scilab.

```
--> y=octave_fun("SIG",1)
error: Invalid call to SIG.  Correct usage is:

 -- SIG ()

Additional help for built-in functions and operators is
available in the online version of the manual.  Use the command
'doc <topic>' to search the manual index.

Help and information about Octave is also available on the WWW
at https://www.octave.org and via the help@octave.org
mailing list.

Error from Octave
```

Figure 3.3: Error from Octave inside Scilab

3.2 Task 2: Structure Support

3.2.1 Problem Statement

Handling octave functions that represent the input and output data in structure format.

The toolbox before this task was supporting doubles/ complex numbers and strings but no structured I/O. Upon receiving a struct as an input or output the toolbox simply gave a "unsupported datatype" error. Our task was to extend the toolbox to accept and return structs as well.

```
--> s=struct('date',13,'month','JAN')
 s  =

  date: [1x1 constant]
  month: [1x1 string]


--> s=octave_fun("setfield",s,'year',2021)

octave_fun: Wrong type of input argument 1.
```

Figure 3.4: Original Struct I/O Error

3.2.2 Role & Contributions

Role: Developer

This task was a bit trickier as we were dealing with not one but different types of data inside a structure. The basic approach to this problem that we used is to fetch the data from Scilab, store it in a custom defined C++ structure, interface with Octave, get the returned data stored in the same custom structure and finally deliver it to Scilab for the user.

Problems Faced:

1. Transferring data between Octave and Scilab's API.
2. Storing the data between the two APIs.
3. Handling of heterogeneous data inside a structure.

3.2.2.1 Transfering Data Between Scilab and Octave API

The first hurdle to cross was to get the struct data from Octave and Scilab's API whether it is input or output. Since a struct is nothing but a mapping of various key value pairs, we needed to get the keys and their respective values.

In Octave we extensively made use of the `octave_scalar_map` class and it's member to create structures and send them as I/O while using Octave's C++ API.

For **fetching** the data, in Scilab we used the `scilab_getFields` function and appropriate similar functions to get the respective value pair depending on the datatype. In Octave, we made use of the `octave_scalar_map` class and it's function to extract the struct data, most importantly the `key` and `contents` members for the keys and values respectively.

On the other hand for **sending** the data to the API, In Octave we we used the `assign` member of `octave_scalar_map` to assign the key-value pairs of a struct and in Scilab, we used `scilab_createStruct` to model a structure. To assign a key to this Scilab structure `scilab_addField` and it's equivalent counterpart based on the datatype was used.

```
else if(out(ii).isstruct()){
    inp[ii].is_out_struct = 1;

    octave_scalar_map outOctaveStruct = out(ii).scalar_map_value();

    int structLen = outOctaveStruct.nfields();
    inp[ii].n_out_struct_len = structLen;

    inp[ii].out_struct = (FUNCSTRUCT *) malloc(sizeof(FUNCSTRUCT) * structLen);
    FUNCSTRUCT* outStruct = inp[ii].out_struct;

    octave_scalar_map::iterator idx = outOctaveStruct.begin();
    int j = 0;

    // std::cout << "data in fun.cpp\n";
    // populating structure
    while (idx != outOctaveStruct.end()){
        std::string currKey = outOctaveStruct.key(idx);
        octave_value currValue = outOctaveStruct.contents(idx);
```

Figure 3.5: Fetching Struct Data from Octave

```
else if (scilab_getType(env, in[i]) == 18) //Checking for Struct input
{
    ins[i].type = TYPE_STRUCT;
    wchar_t** keys = NULL;
    scilabVar struct_out;
    int dims = 0;

    // call getfields only when struct is not empty else getfields will crash
    if(scilab_isEmpty(env, in[i]) != 1){
        dims = scilab_getFields(env, in[i], &keys); // Retrieving Struct Keys
    }
    ins[i].n_in_struct_len = dims;
    //std::cout<<dims<<std::endl;

    // allocating memory for keys and values
    ins[i].in_struct = (FUNCSTRUCT*) malloc(sizeof(FUNCSTRUCT) * dims);
    FUNCSTRUCT* inStruct = ins[i].in_struct;

    for (j = 0; j < dims; j++)
    {
        // storing the key
        inStruct[j].key = malloc(sizeof(wchar_t) * (wcslen(keys[j]) + 1));
        wcscpy((wchar_t*) inStruct[j].key, keys[j]);

        struct_out = scilab_getStructMatrix2dData(env, in[i], keys[j], 0, 0); // Retrieving Curr Value
```

Figure 3.6: Fetching Struct Data from Scilab

Since the value of each field in as struct could be a different data type we had to

keep a tight conditional check on extracting/ storing these different types of data types. Right now the structs can handle doubles, complex numbers, strings and matrices.

3.2.2.2 Storing the data between the two APIs.

The next hurdle was to store the data in between the two APIs, the main issue being since the struct can contain heterogeneous data, traditional arrays would not do.

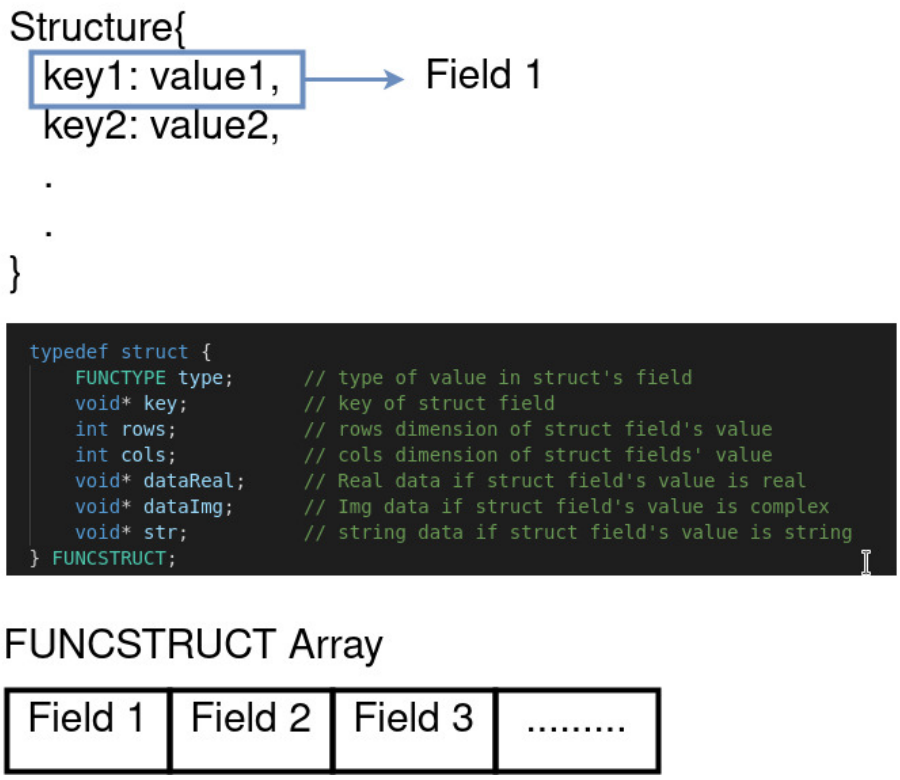


Figure 3.7: Custom Structure to Store Data

The solution that I came up with was to use a custom defined C++ structure for this purpose. We would create an array of this custom structure and each index of this array would replicate each field of the struct that we are transferring between the API. Each field or index would store field as well as it's respective value. This way we could recreate the structure for Scilab or Octave as well and it is flexible enough to support any custom type of user created structs.

The custom structure used to store data is of the form:

```
typedef enum {
    TYPE_DOUBLE,
    TYPE_COMPLEX,
    TYPE_STRING,
```

```
    TYPE_STRUCT,
}FUNCTYPE;

typedef struct {
    FUNCTYPE type;    //Type of value in struct's field
    void* key;        //key of struct field
    int rows;         //rows dimension of struct field's value
    int cols;         //cols dimension of struct fields' value
    void* dataReal;   //Real data if struct field's value is real
    void* dataImg;    //Img data if struct field's value is complex
    void* str;        //String data if struct field's value is string
} FUNCSTRUCT;
```

3.2.2.3 Additional Task

One more thing that we added to the toolbox that was not explicit to this task was string I/O compatibility.

3.2.3 Conclusion

Now the toolbox supports struct I/O and the structs can contain doubles, complex numbers, strings and matrices. Additionally the toolbox now also supports string I/O.

```
--> y=octave_fun("SIG")
 y  =

  ABRT = 22
  FPE = 8
  ILL = 4
  INT = 2
  SEGV = 11
  TERM = 15
```

Figure 3.8: Working Struct I/O in Scilab Octave Toolbox

3.3 Task 3: Windows OS Support

3.3.1 Problem Statement

Extending the toolbox usage to Windows OS.

The Original Scilab Octave Toolbox was only supported on Linux. The third task was to bring the toolbox support for Windows OS as well.

3.3.2 Role & Contributions

Role: Manager

Adding support for Windows was faced with some unique hurdles as the loading and building process is quite differ net than on Linux.

As the manager for this task I divided the problem into three main parts:

- Creating the builder script.
- Creating the loader script.
- Testing and Fixing the bugs on Windows OS.

Problems Faced

1. Loader script failing without bin PATH.
2. Octave Interpreter failing to search for function files without "OCTAVE_HOME" environment variable.
3. Cleaner script not working due to multiple gcc binaries in the PATH.
4. Crashes while running the test script in Scilab.

```
--> exec loader.sce

--> // This file is released under the 3-clause BSD license. See COPYING-BSD.

--> // Generated by builder.sce: Please, do not edit this file


--> oldmode = mode();      mode(-1);
Start scilab_octave
        Load macros
        Load gateways
at line    15 of executed file E:\Karan\dev\FOSEE\fossee-scilab-octave-toolbox\loader.sce

exec: error on line #18: "link: The shared archive was not loaded: Unknown Error"
```

Figure 3.9: loader.sce error in Scilab on Windows OS

3.3.2.1 Handling the Errors

The solution to this task was the result of constant debugging and going through the documentation was creating toolboxes on Scilab. The most time consuming part was to getting to the source of various error in the building and loading process and random crashes.

The error of the builder script failing was tracked back to the presence of multiple `gcc` binaries in the PATH environment variable. We need the `gcc` version specified by the `mingw toolbox` but the Octave installation also ships with it's own `gcc` executable and having that in the PATH would mean trouble for the builder script. Simply removing the `octave/bin` from PATH while building the toolbox would solve this problem.

Coming to the loader script, it would fail if we do not have the `octave/bin`in our PATH variable (which we removed for the build process). Thus we need to add that back to the PATH variable. Since building is a one time process we could add back the bin path.

After successfully loading the toolbox into Scilab we then encountered a unique problem where Octave's interpreter was not able to find functions. This was traced to the absence of the `OCTAVE_HOME` user variable, which is essential for Octave's interpreter to find function definitions.

3.3.3 Conclusion

We achieved the support by modifying the builder script and adding the equivalent prerequisites for Windows.

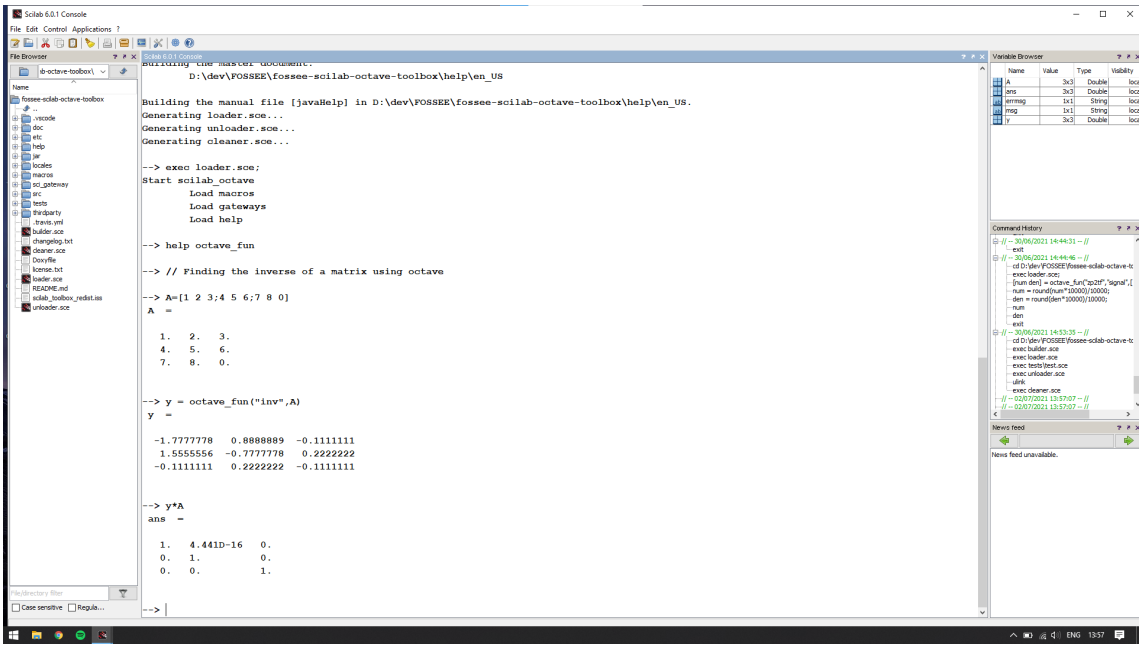


Figure 3.10: Scilab Octave Toolbox working on Windows OS

Chapter 4

Conclusion

Closing off this report, the Scilab Octave Toolbox at the end of this Semester Long Internship is now capable of:

- Handling and Displaying Error Messages from Octave.
- String Input/Output
- Structured Data as Input/Output
- Running on Windows OS

During the semester-long internship, I got the chance to work on open-source software and even though sometimes the learning curve was a bit steep it proved to be a great opportunity.

At the end, I would like to thank the mentors, my team members and everyone who made my internship a great learning experience.

Chapter 5

Reference

- Source Repository:
  https://github.com/FOSSEE/fossee-scilab-octave-toolbox
- Scilab API Documentation:
  https://help.scilab.org/docs/6.0.2/en_US/api_struct.html
- Octave API Documentation:
  https://octave.org/doxygen/4.4/
- https://octave.1599824.n4.nabble.com/Capturing-exception-messages-in-octave-C-API-td4694132.html