Semester Long Internship Report

On

Scilab Signal Processing Toolbox development

Submitted by

Abinash Singh

Under the guidance of

Prof.Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

Mentor

Ms. Rashmi Patankar

September 4, 2025

Acknowledgment

I am deeply grateful to my mentor, Ms. Rashmi Patankar, for her invaluable guidance, support, and encouragement throughout my journey with the FOSSEE Team at IIT Bombay.

I began as a Semester-Long summer Intern in 2024, and this report marks the completion of my Semester long Winter Internship (2024). This experience has been enriching, allowing me to enhance my technical skills further and contribute meaningfully to open-source development.

I would also like to sincerely thank Prof. Kannan M. Moudgalya and Prof. Kumar Appaih for their insightful guidance, which has significantly shaped my understanding of open-source systems.

Additionally, I extend my heartfelt gratitude to my friends who supported me in completing the screening tasks for this internship. Their encouragement and assistance were crucial in overcoming challenges and achieving key milestones.

I remain committed to contributing to Scilab and other FOSSEE initiatives. In the coming months, my primary focus will be on advancing the Signal Processing Toolkit, and I look forward to making further meaningful contributions.

I am thankful to everyone who has been part of this journey, offering their support and inspiration along the way. Your unwavering belief in my abilities has made this experience truly fulfilling.

Contents

Chapter 1

Introduction

Scilab is a free and open-source, cross-platform numerical computational package and a high-level, numerically oriented programming language.

It can be used for signal processing, statistical analysis, image enhancement, fluid dynamics simulations, numerical optimization, and modeling, simulation of explicit and implicit dynamical systems, and (if the corresponding toolbox is installed) symbolic manipulations.

Scilab is one of the two major open-source alternatives to MATLAB, the other one is GNU Octave. Scilab puts less emphasis on syntactic compatibility with MATLAB than Octave does, but is similar enough to easily transfer skills between the two systems.

IIT Bombay is leading the effort to popularise Scilab in India and the Scilab Signal Processing Toolbox is one of its endeavors toward the cause. This effort is part of the Free and open-source Software for Science and Engineering Education (FOSSEE) project, supported by the National Mission on Education through ICT of the Ministry of Education.

FOSSEE Scilab Signal Processing Toolbox is a comprehensive suite designed for the analysis, manipulation, and visualization of signals, developed and maintained by FOSSEE, IIT Bombay.

It offers a wide range of functions that cover fundamental and advanced signal processing techniques, including filtering, spectral analysis, and time-frequency analysis.

Users can implement various types of digital filters (such as FIR and IIR), perform Fourier transforms, and analyze the frequency content of signals. The toolbox also supports wavelet transform methods, which are essential for non-stationary signal analysis.

With its intuitive interface and extensive testing, the Signal Processing Toolbox is a powerful tool for engineers, researchers, and ed- educators working in fields like telecommunications, audio processing, and biomedical signal analysis.

Chapter 2

Signal Processing Toolbox Development

2.1 Overview

The **FOSSEE Signal Processing Toolbox for Scilab** is a comprehensive collection of functions that enable users to perform a wide range of signal processing tasks such as filtering, transforms, spectral analysis, convolution, and correlation directly within Scilab. It is primarily intended for students, researchers, and professionals who wish to carry out signal processing computations using an open-source platform without depending on proprietary tools.

The toolbox is structured as a set of macros located under the directory `FOSSEE-SignalProcessingToolbox/macros/`. In earlier versions, these macros fell into two categories:

1. **Scilab-native functions:** Implemented completely in Scilab, performing all computations internally.
2. **FSOT-based wrapper functions:** Depended on the FOSSEE Scilab–Octave Toolbox (FSOT) to call Octave's signal processing functions, with Scilab serving only as an interface.

In older versions, a significant portion of the toolbox relied on FSOT-based wrappers. However, this design introduced several drawbacks, including:

- **Performance overhead:** Communication between Scilab and Octave slowed down execution.
- **Additional dependencies:** Users needed both FSOT and Octave installed, complicating setup and portability.
- **Limited functionality:** FSOT lacked support for certain data types and operations, such as boolean values, structs, and graphical/image data.

As part of this internship project, the entire codebase was refactored. **All FSOT-based functions were re-implemented as Scilab-native functions.** As a result, the toolbox is now completely independent of FSOT and Octave. Every

function executes natively within Scilab, making the toolbox more reliable, efficient, and easier to maintain.

The key motivations for this refactoring were:

- **Complete elimination of dependencies:** The toolbox now runs entirely within Scilab without requiring FSOT or Octave.
- **Improved performance:** Direct execution in Scilab avoids the overhead of inter-language communication.
- **Expanded functionality:** Functions are no longer constrained by FSOT's limitations, allowing support for a wider range of use cases.
- **Ease of maintenance:** A fully Scilab-native codebase simplifies debugging, testing, and future extensions.

Through this refactoring, the toolbox has evolved from a hybrid Scilab–Octave system into a fully self-contained, Scilab-native package. The following sections describe the methodology adopted for this conversion and the steps undertaken to ensure correctness, efficiency, and feature completeness.

2.2 Development Workflow

The development workflow for refactoring the toolbox evolved significantly over the course of the internship as experience and familiarity with both Octave and Scilab increased. Initially, the approach was more direct and mechanical, but over time it matured into a systematic process that ensured correctness, efficiency, and completeness. The workflow can be summarized in the following stages:

1. **Studying the Octave implementation:** Each function to be ported was first studied in detail by examining its Octave source code. This helped in understanding the underlying algorithm, the expected inputs and outputs, and any special cases or edge conditions handled in the original implementation.
2. **Line-by-line translation into Scilab:** The Octave code was then translated into Scilab syntax and constructs. In the initial stages, this process was often a direct linebyline conversion, which served as a baseline to replicate the functionality in Scilab.
3. **Identifying and implementing missing functions:** In many cases, Octave relied on utility functions or built-in routines that were not directly available in Scilab. Such functions had to be implemented separately in Scilab to achieve functional parity. This step required both algorithmic understanding and careful coding to ensure efficiency.
4. **Testing, debugging, and iterative refinement:** After translation, the newly written Scilab functions were tested against known inputs and compared with the outputs from Octave implementations. Any discrepancies were

debugged, and the functions were refined iteratively until results matched expected behavior. This stage also involved optimizing the code for performance where necessary.

Over time, this workflow shifted from a basic translation process to a more *designoriented approach*, where the focus was not merely on replicating Octave code but on producing robust, Scilabnative implementations that integrate seamlessly into the toolbox.

2.2.1 Reading Octave Implementation

Analyzing the existing Octave implementation is the first and most crucial step in the workflow. This stage establishes the foundation for the translation process by helping us understand the algorithm, dependencies, and complexity of the function under consideration. It is particularly important to note that certain helper or sub-functions used in Octave may not have direct equivalents in Scilab, which makes this step essential for planning the subsequent development phases.
To locate and study the Octave source code of a given function, one can typically refer to two main sources:

1. **Octave Signal Package:** Functions related to signal processing are generally part of Octave's `signal` package. The source code for this package is publicly available at: https://octave.sourceforge.io/signal/ Most of these functions are implemented directly in Octave (using `.m` files), although some are written in C++ for performance reasons.

2. **Octave Core Package:** If a function does not belong to the signal package, it is usually part of Octave's core distribution. In such cases, the source code can be obtained by downloading the latest Octave release and searching for the function in the extracted source tree.

The outcomes of this analysis are threefold:

- **Estimation of effort and time:** Understanding the implementation provides a rough estimate of the time required for translation.

- **Identification of missing sub-functions:** Any sub-functions or utilities unavailable in Scilab can be listed for separate implementation.

- **Assessment of complexity:** By studying the structure of the Octave code, we can anticipate challenges in the translation process and plan accordingly.

Overall, this stage ensures that the translation process begins with a clear understanding of both the functional requirements and the potential gaps between Octave and Scilab environments.

2.2.2 Line-by-Line Translation

Once the Octave implementation has been studied, the next step involves translating the code into Scilab. This process is carried out in a meticulous line-by-line manner to ensure that the translated function behaves identically to its Octave counterpart.
The primary focus during this stage is on identifying and addressing syntax-level differences between Octave and Scilab.
Although the two languages share a similar structure, several subtle differences must be taken into account. Some of the key distinctions are as follows:

- **Control flow statements:** In Octave, conditional and loop constructs require explicit terminators. For example, `if`, `else`, `for`, `while`, and `switch` statements must be closed with `endif`, `endfor`, `endwhile`, and `endswitch`, respectively. In Scilab, a simple `end` is sufficient to terminate all such constructs.
- **Constants:** Certain constants are represented differently in the two languages. For example:
  - Octave: `pi`, `eps`, `j`, `true`, `false`
  - Scilab: `%pi`, `%eps`, `%i`, `%T`, `%F`
- **Error handling:** Octave provides the `print_usage` function for handling incorrect function usage. Scilab does not support this; instead, an equivalent error can be raised using `error("message")`.
- **General syntax similarities:** Beyond these differences, the overall syntax and function structure in Octave and Scilab are largely similar, which makes the translation process relatively straightforward once these distinctions are understood.

Careful attention to these differences ensures that the translated code is both functionally accurate and idiomatic to Scilab, reducing the likelihood of runtime errors and improving maintainability.

2.2.3 Comparing In-Built Functions and Writing Missing Ones

After identifying which sub-functions are available in Octave and verifying their counterparts in Scilab, you may still encounter discrepancies—not just in availability but also in behavior. Even when functions share the same name across platforms, their semantics or implementation details can differ.

To accurately compare or substitute functions, always rely on the official documentation. Scilab provides a dedicated "Matlab–Scilab equivalents" section—your go-to reference for mapping Matlab/Octave functions to their Scilab counterparts.
If a Scilab equivalent behaves differently or does not exist, there are two practical approaches:

- Write a **wrapper** in Scilab to normalize behavior.
- **Re-implement** the algorithm in Scilab for accuracy and maintainability.

Sample Octave/MATLAB vs Scilab Mappings *(Examples adapted from the official mapping documentation)*

Octave / MATLAB	Scilab Equivalent
`abs(a)`	`abs(a)` — absolute value (same semantics)
`cell(...)`	Use Scilab lists or tlist structures (e.g., `list()`, `tlist()`, `tlstcopy()`)
`string(a)`	`string(a)` —
`all(a)`	`all(a)` — consistent semantics between platforms
`length(a)`	`max(size(a))` — Scilab doesn't have `length()` by default
`end` index alias	Scilab uses `$` as the equivalent of `end` indexing

Behavioral Differences to Watch Out For

- `max` / `min` usage: In Octave/MATLAB, `max(A)` returns maximum values column-wise, whereas in Scilab it may operate over the entire matrix by default—so be explicit if replicating MATLAB's behavior.

Workflow for Missing or Divergent Functions

1. Review Matlab/Octave documentation to determine the expected behavior.
2. If possible, inspect the source code (e.g., from Octave) to understand implementation details.
3. Implement equivalent logic in Scilab with a focus on clarity and performance.

Handling C++-Based Implementations

If the original functionality is implemented in C++:

- **Option 1**: Extract the algorithm and recreate it entirely in Scilabthis keeps maintenance simpler and language-consistent.
- **Option 2**: Use Scilab's external interface or C++ API to wrap the compiled code—this retains original performance but adds complexity and potential portability challenges.

Unless you are confident in C++ extension development, **Option 1** is usually more sustainable and less error-prone.

2.2.4 Test And Iterate

This is a crucial but often tedious part of the workflow. To lighten the load, you can find test cases at the bottom of the Octave source code files. Attempt these first, and if you encounter difficulties running some test cases, don't worry—create your own examples instead.

Compare the outputs of these examples with those from Octave and your implemented function. Ensure to write diverse test cases covering different calling sequences, aiming for at least one example for each type.

In addition to these basic checks, more rigorous testing is required. Simple functional verification is not sufficient; you should also perform **regression tests** to ensure that future changes do not break existing functionality. Stress tests with edge cases and large inputs are equally important to validate both correctness and performance under heavy workloads.

A practical way to manage tests is to include them directly in your source files as comments at the end. For example:

```
# Test cases
# assert( myfunc(1, 2), 3 )
# assert( myfunc([1 2 3]), [1 2 3] )
# assert( isequal(myfunc(zeros(3,3)), zeros(3,3)) )
```

Here, each test should be written using `assert` along with `isequal` (when dealing with arrays or matrices). Always provide the expected result explicitly so that both you and future contributors can immediately see the intended behavior. This also serves as lightweight documentation for usage examples.

2.3 Previous Status

Following the previously outlined workflow, I successfully translated the following during my semester-long fellowship in summer of 2024 :

- 27 functions independently.
- 2 functions in collaboration with another intern.

In total, our team had translated approximately 60 functions.
Each function is accompanied by documentation at the top of its file and test cases at the bottom.

2.4 Current Status

My Semester Long Internship in winter of 2024 with FOSSEE continued this work and achieved the following:

- 19 more functions are ported from octave to Scilab.
- assigned issues and to do to 16 functions that I was not able to port during the internship.

2.4.1 Documentation pattern

Since the functions are intended to mirror their implementation in Octave, it's prudent to use Octave's documentation as a reference for documenting the corresponding Scilab functions.

Additionally, consult Matlab's documentation for these functions, as Octave may have certain bugs. If your test cases fail despite following all steps, compare your code's output with Matlab's to troubleshoot further.

The documentation for a function sits just below the function's declaration, and is written as one big comment block. There are four components to a function's documentation:

1. Syntax: The order of evaluation for the function for the set of given parameters.
2. Parameters: The expected values for the function's parameters. This section outlines the type as well as the acceptable values for these parameters.
3. Description: A comprehensive description of the function. This section outlines default behaviors, expected input and output, and how to interpret them, dependencies and other such data
4. Examples: An example to demonstrate the correct usage of the function.

It is worth noting that in the process of re-writing functions to use Scilab code, the documentation part does not require a lot of modifications because the intended behavior for the function is, most of the time, already well-documented and in line with their Octave counterparts.

All of my work is available in my GitHub repository GITHUB

2.4.2 Functions Completed During Semester Long Internship summer 2024

S.No	Function	Dependencies/custom functions
1	fftn	
2	fht	
3	fft1	
4	fft21	
5	fftconv	
6	ifft1	
7	ifft2	
8	ifftn	
9	idct2	
10	idct1	idct1
11	idst1	dst1
12	czt	fft1 , ifft1
13	xcorr2	
14	shanwavf	
15	rceps	fft1 , ifft1
16	pulstran	
17	hilbert1	fft1 , ifft1 , ipermute
18	grpdelay	fft1
19	pwelch	fft1
20	tfe	pwelch
21	mscohere	pwelch
22	cpsd	pwelch
23	cohere	pwelch
24	arch_fit	autoreg_matrix , ols
25	arch_test	ols
26	spectral_xdf	fft1
27	spectral_adf	ifft1 , fft1
28	unwrap2	ipermute
39	cplxreal	cplxpair , ipermute

2.4.3 Functions Completed During Semester Long Internship Winter 2024

S.No	Function	Dependencies/custom functions
1	besself	besselap bilinear postpad prepad sftrans tf2zp zp2tf
2	bitrevorder	digitrevorder
3	cell2sos	
4	sos2cell	
5	cummax	
6	cummin	
7	ellip	ellipap sftrans zp2tf
8	ncauer	ellipap
9	periodgram	fft1 hamming
10	impz	fftfilt
11	iirlp2mb	
12	marcumq	
13	freqz	fft1 postpad unwrap2
14	invfreq	
15	invfreqz	invfreq
16	invfreqs	invfreq
17	findpeaks	
18	impinvar	deconv residue
19	invimpinvar	impinvar

2.4.4 issues and possible Improvements

- **pwelch** : (semilogx,semilogy,etc) and other plot functions can't handle negative values
- **arch_fit and arch_test** : singular matrix exception handling can be improved using try-catch statements.
- There is some bug in the argn() function because of which we get 1 even if our output args are zero. Temporary fix: functions with plot will plot if there is 1 output arg.
- Documentation can be improved with practical examples

2.5 Scilab–Octave Mapping of Functions

One of the main goals of this internship was to study how Octave functions can be ported to Scilab.The table below shows the final mapping of toolbox functions:

Scilab Function	Octave Function
fftshift1	fftshift
sinetone	sinetone
sinewave	sinewave
fftn	fftn
fht	fht
filtfilt	filtfilt
gaussian	gaussian
triang	triang
filter2	filter2
fft1	fft
fft21	fft2
fftconv	fftconv
ifft1	ifft
ifft2	ifft2
ifftn	ifftn
morlet	morlet
durbinlevinson	durbinlevinson
hurst	hurst
polystab	polystab
dst1	dst
autoreg_matrix	autoreg_matrix
cceps	cceps
idct2	idct2
idct1	idct
idst1	idst
clustersegment	clustersegment
cplxreal	cplxreal
detrend1	detrend
ifftshift1	ifftshift
tripuls	tripuls
unwrap2	unwrap
ifht	ifht
arma_rnd	arma_rnd
pei_tseng_notch	pei_tseng_notch
czt	czt
xcorr2	xcorr2
synthesis	synthesis
shanwavf	shanwavf
rceps	rceps
pulstran	pulstran
hilbert1	hilbert
grpdelay	grpdelay
spencer	spencer
stft	stft
schtrig	schtrig
tf2sos	tf2sos
tfestimate	tfestimate
ar_psd	ar_psd
pwelch	pwelch

Scilab Function	**Octave Equivalent**
tfe	tfe
mscohere	mscohere
cpsd	cpsd
cohere	cohere
arch_fit	arch_fit
arch_test	arch_test
fwhmjlt	fwhm
pburg	pburg
fwhm	fwhm
spectral_xdf	spectral_xdf
spectral_adf	spectral_adf
ols	ols
cplxpair	cplxpair
ipermute	ipermute
zp2sos	zp2sos
besself	besself
bitrevorder	bitrevorder
cell2sos	cell2sos
sos2cell	sos2cell
cummax	cummax
cummin	cummin
ellip	ellip
ncauer	ncauer
periodogram	periodogram
impz	impz
iirlp2mb	iirlp2mb
marcumq	marcumq
freqz	freqz
invfreq	invfreq
invfreqz	invfreqz
invfreqs	invfreqs
findpeaks	findpeaks
impinvar	impinvar
invimpinvar	invimpinvar

This mapping provides a quick reference for comparing Scilab and Octave functions. It will be useful for anyone working on code migration, validation, or further development of the toolbox.

Chapter 3

Toolbox Documentation in Scilab

Documentation plays a very important role in any software system. If there is a lack of documentation for users, it will not be easy for them to use the software effectively. At the same time, if a system lacks developer documentation, it becomes difficult to maintain and update. Documentation should never be neglected, and we should dedicate significant effort to creating it properly.

3.1 Toolbox Structure in Scilab

Before documenting a toolbox, it is essential to understand its internal structure. Scilab toolboxes follow a well-defined directory and file convention, enforced by the ATOMS packaging system. All official toolboxes uploaded to ATOMS must conform to this structure. A ready-to-use template, called the `toolbox_skeleton`, is distributed with Scilab to serve as a starting point.

3.1.1 Directory Layout

A typical Scilab toolbox has the following directories:

- **macros** – Contains user-level Scilab functions (`.sci` files) and a builder script (`buildmacros.sce`). This folder is *mandatory*.
- **src** – Native source code (`.c`, `.cpp`, `.f`, etc.), along with builders for compiling these sources. Present only if the toolbox provides primitives.
- **sci_gateway** – Gateway functions (`.c`, `.cpp`, etc.) wrapping native code. Only needed if primitives exist.
- **help** – Documentation files in XML format (organized per language, e.g. `en_US`, `fr_FR`). Strongly recommended.
- **etc** – Initialization and cleanup scripts (`<toolbox>.start`, `<toolbox>.quit`). The `.start` file is mandatory.
- **tests** – Regression and unit test scripts (`.tst`), highly encouraged.

- **demos** – Usage examples (`.sce`), optional but useful for users.
- **includes** – Public header files (`.h`), optional.
- **jar** – Generated Java packages or help JARs (created at build time).
- **stagging** – Incomplete functions not ready for release.

3.1.2 Key Files

At the root of the toolbox, a few important files are expected:

- `builder.sce` – The main build script, which coordinates sub-builders.
- `loader.sce` – Generated loader script for manually loading the toolbox.
- `DESCRIPTION` – Metadata about the toolbox (name, version, author, dependencies).
- `readme.txt`, `license.txt` – Optional but strongly recommended.
- `test.sce`, - Main root level test script.

3.1.3 Lifecycle: Build and Load

The toolbox is first compiled and packaged using:

```
exec builder.sce
```

This generates the necessary binaries, libraries, and loaders. It can then be loaded in Scilab with:

```
exec loader.sce
```

At Scilab startup, the `etc/<toolbox>.start` script ensures the toolbox is automatically initialized, and the `etc/<toolbox>.quit` script is executed at shutdown.

This modular design separates user-facing macros, native code, gateways, documentation, and tests. It provides a clean foundation for writing both developer documentation and user-facing manuals, ensuring the toolbox is easy to maintain, extend, and distribute.

3.2 Documentation Types in Scilab

Scilab supports two distinct types of documentation, each serving a different purpose:

1. **User Documentation** - Generated from function headers, accessible through Scilab's help system
2. **Developer Documentation** - In-code comments that explain implementation details for developers

This chapter focuses primarily on the user documentation generated from function headers.

3.3 User Documentation Through Function Headers

3.3.1 The Function Header Comment Structure

Scilab automatically generates user documentation from specially formatted comments in function headers. These comments follow a structured format that enables Scilab to parse them into comprehensive help documentation:

```
function [z]=function_name(x, y)
// Short description on the first line following the function header.
//
// Syntax
//   [z] = function_name(x,y) // calling examples, one per line
//
// Parameters
// x: the x parameter   // parameter name and description must be
// y: the y parameter   // separated by ":"
// z: the z parameter
//
// Description
// Here is a detailed description of the function.
// Add an empty comment line to format the text into separate paragraphs.
//
// Examples
// [z] = function_name(1, 2) // examples of use
//
// See also
//  related_function1
//  related_function2
//
// Authors
//  Author name ; additional information
//
// Bibliography
//  Reference information

// Actual function code starts here
z = sin(x).*cos(x + y);
endfunction
```

3.3.2 Importance of Auto-Generated Documentation

The auto-generation of documentation from function headers offers several significant advantages:

1. **Single Source of Truth** - Documentation lives with the code, reducing discrepancies

2. **Automatic Updates** - When function parameters or behavior change, updating the header comments automatically updates the documentation

3. **Consistency** - Enforces a standard documentation format across all functions

4. **Integration with Help System** - Automatically becomes accessible through Scilab's help command

5. **Reduced Documentation Burden** - Developers only need to maintain one set of documentation

3.3.3 XML Files: The Critical Intermediate Format

The function header comments are automatically converted to XML files, which serve as an intermediate representation before final compilation into JAR files. This conversion is crucial because:

1. **Structured Format** - XML provides a well-defined structure for documentation

2. **Rich Content Support** - XML allows for formatting, equations, lists, and other rich content

3. **Cross-referencing** - Enables linking between related functions

4. **Processing Capabilities** - Can be transformed into various output formats

5. **Internationalization Support** - Facilitates translation into multiple languages

3.3.4 From XML to JAR: Final Documentation Format

The XML files are compiled into JAR files, which:

1. **Integrate with Scilab's Help System** - Making documentation accessible via the `help` command

2. **Provide Efficient Storage** - Compressing documentation for distribution

3. **Enable Indexing and Searching** - For quick access to specific documentation

4. **Support Rich Media** - Including formatted text, equations, and diagrams

3.4 The Documentation Generation Process

The process of generating user documentation from function headers involves:

1. **Writing Header Comments** - Following the standard structure in function files
2. **Running help_from_sci** - To extract and convert header comments to XML

   ```
   help_from_sci("function_name.sci", "en_US");
   ```

3. **Building the Help System** - Compiling XML into JAR files

   ```
   tbx_builder_help(TOOLBOX_TITLE);
   ```

4. **Accessing Documentation** - Users can then access help via:

   ```
   help function_name
   ```

The document generation is usually controlled by the toolbox build script. After loading or installing the toolbox, users can simply access documentation by typing "help function_name".

3.5 Developer Documentation vs. User Documentation

While user documentation is generated from function headers, developer documentation consists of in-code comments that explain implementation details:

3.5.1 Developer Documentation Best Practices

```
function [z]=function_name(x, y)
// USER DOCUMENTATION HEADER COMMENTS HERE
// ...

// DEVELOPER DOCUMENTATION STARTS BELOW

// This function computes the product of sine and cosine
// Uses element-wise multiplication for better performance
z = sin(x).*cos(x + y);

// Note: Input arguments are not checked for consistency
// TODO: Add input validation in future versions

endfunction
```

Key differences between user and developer documentation:

User Documentation (Function Headers)	Developer Documentation (In-code Comments)
Focuses on usage and interface	Focuses on implementation details
Structured format for help system	Free-form comments within code
Accessible through `help` command	Only visible when reading source code
Describes what the function does	Explains how and why it does it
Includes examples and parameters	Includes rationale and technical notes

Table 3.1: Comparison of User and Developer Documentation

3.6 Best Practices for Function Header Documentation

To create effective user documentation from function headers:

1. **Be comprehensive** - Include all standard sections
2. **Provide clear examples** - Show typical usage patterns
3. **Document all parameters** - Clearly describe inputs and outputs
4. **Use proper formatting** - Utilize XML tags for enhanced formatting
5. **Include cross-references** - Link to related functions
6. **Update headers when code changes** - Keep documentation synchronized with implementation

3.7 Documentation in Signal Processing Toolbox

The `builder.sce` script is responsible for building the toolbox:

```
// In builder.sce
tbx_builder_help(toolbox_dir);
```

This line will build the help by executing any file with a name beginning with "build" inside the `help/` directory.

We have a `builder_help.sce` file in the help folder:

```
// In builder_help.sce
help_from_sci(toolbox_dir+"/macros/", help_dir+"/en_US/");
```

This line is responsible for generating XML files from function header comments.

Then `builder_help.sce` calls `en_US/build_help.sce` script to continue the process.

There is a consistent pattern for building documentation in various toolboxes, making it easy to follow and implement in your own toolboxes.

3.8 Conclusion

The function header documentation system in Scilab provides a powerful mechanism for maintaining synchronized, accessible, and comprehensive user documentation. By embedding documentation directly in function headers and automatically converting it to XML and JAR formats, Scilab ensures that users always have access to up-to-date information on function usage.

This approach bridges the gap between code and documentation, reducing maintenance burden while improving documentation quality. When combined with proper developer documentation in the form of in-code comments, Scilab toolboxes can be both user-friendly and maintainable from a development perspective.

Remember that effective documentation is a key factor in the adoption and proper use of your toolbox. Investing time in creating thorough function header documentation pays dividends in user satisfaction and reduced support requirements.

Chapter 4

Publishing a Scilab Toolbox on ATOMS

Introduction

ATOMS (AuTomatic mOdules Management for Scilab) is Scilab's package management system. It allows developers to distribute external modules (toolboxes) to the community, while enabling users to easily install, update, and manage them directly through the Scilab interface or command line.

This chapter provides a complete guide on how to:

- Structure a Scilab toolbox
- Write builder and loader scripts
- Add macros, primitives, and Java code
- Create documentation and help files
- Package and submit your toolbox to the ATOMS portal

The process relies on the `toolbox_skeleton`, a template module provided with Scilab (available under `SCI/contrib/toolbox_skeleton`). This skeleton demonstrates the directory structure and minimal files needed.

4.1 General Concepts

Before creating a toolbox, it is essential to understand the core terminology:

Script: A Scilab code file with `.sce` extension, executed with `exec()`.

Macro: A Scilab function written in Scilab language, stored in a `.sci` file.

Primitive: A Scilab function implemented in a compiled language (C, C++, Fortran). Requires a gateway.

Gateway: A C/C++ function that connects Scilab to native functions, converting inputs/outputs.

Builder: A script (`builder.sce`) used to build and compile a toolbox into a loadable form.

Loader: A script (`loader.sce`) used to load the toolbox into Scilab.

Start/Quit Scripts: Scripts in the `etc/` directory, executed when Scilab starts/quits the toolbox.

4.2 Toolbox Structure

By convention, all ATOMS toolboxes follow a defined structure. The root directory is named after your toolbox and contains the following:

Directories

Directory	Contents	Required?
`macros/`	Scilab macros (`.sci`), macros builder (`buildmacros.sce`)	Yes
`src/`	Source code files (`.c`, `.cpp`, `.f`, ...), builders per language	Only if needed
`sci_gateway/`	Gateway source files (`.c`), gateway builder & loader scripts	Only if primitives exist
`help/`	Documentation in XML, organized by language (`en_US`, `fr_FR`, ...)	Strongly suggested
`etc/`	Initialization (`.start`) and finalization (`.quit`) scripts	Mandatory
`tests/`	Unit and regression tests (`.tst`)	Suggested
`demos/`	Example scripts (`.sce`)	Optional
`includes/`	Public header files (`.h`)	Optional

Files

File	Description	Required?
`builder.sce`	Main builder script for the toolbox	Yes
`loader.sce`	Main loader script (auto-generated)	Yes
`DESCRIPTION`	Metadata file (author, version, license, dependencies)	Yes
`readme.txt`	Usage/installation notes	Optional
`license.txt`	Toolbox license	Optional

4.3 Builder and Loader Scripts

The **builder** compiles and packages your toolbox; the **loader** loads it into Scilab.

Main Builder (`builder.sce`)

Example template:

```
mode(-1);
lines(0);
TOOLBOX_NAME  = "mytoolbox";
TOOLBOX_TITLE = "My Toolbox Example";
toolbox_dir   = get_absolute_file_path("builder.sce");

tbx_builder_macros(toolbox_dir);
tbx_builder_src(toolbox_dir);
tbx_builder_gateway(toolbox_dir);
tbx_builder_help(toolbox_dir);
tbx_build_loader(TOOLBOX_NAME, toolbox_dir);

clear TOOLBOX_NAME TOOLBOX_TITLE toolbox_dir;
```

Main Loader (`loader.sce`)

Delegates loading to the `etc/` start script:

```
exec(get_absolute_file_path("loader.sce") + "etc/mytoolbox.start");
```

4.4 Macros

Macros are Scilab functions. They are built into libraries by `buildmacros.sce`.

Example Macro

```
// File: macros/mysum.sci
function s = mysum(A, B)
  s = A + B;
endfunction
```

Macros Builder

```
// File: macros/buildmacros.sce
tbx_build_macros("mytoolbox", get_absolute_file_path("buildmacros.sce"));
clear tbx_build_macros;
```

4.5 Primitives and Gateways

If your toolbox uses compiled code (C, Fortran, etc.), you must create gateways.

Example C Primitive

`csum.c`:

```
int csum(double *a, double *b, double *c) {
    *c = *a + *b;
    return 0;
}
```

`sci_csum.c` (gateway):

```
#include "api_scilab.h"
int sci_csum(char *fname) {
    // Convert inputs, call csum, return result...
}
```

4.6 Help System

Each function should have a help page in XML format.

Example Help File (`mysum.xml`)

```
<refentry xml:id="mysum" xml:lang="en">
  <refnamediv>
    <refname>mysum</refname>
    <refpurpose>Sum of two numbers</refpurpose>
  </refnamediv>
  <refsynopsisdiv>
    <title>Calling Sequence</title>
    <synopsis>s = mysum(a, b)</synopsis>
  </refsynopsisdiv>
  <refsection><title>Description</title>
    <para>Adds two numbers.</para>
  </refsection>
</refentry>
```

Build Help

```
// File: help/build_help.sce
tbx_build_help("My Toolbox Example", get_absolute_file_path("build_help.s
```

4.7 Testing the Toolbox Locally

Run inside Scilab:

```
exec builder.sce
exec loader.sce
```

If successful, your functions are available.

4.8 Submitting to ATOMS

1. Archive your toolbox directory (`.zip` or `.tar.gz`) without compiled binaries.
2. Go to https://atoms.scilab.org/add
3. Log in or create an account
4. Upload your archive and fill metadata (name, version, supported Scilab version, summary, license).
5. After review, your module will be compiled for multiple platforms and published.

4.9 Installing a Published Toolbox

Once online, users can install with:

```
atomsInstall("mytoolbox")
```

Useful Links

- ATOMS Main Portal: https://atoms.scilab.org
- Toolbox Submission: https://atoms.scilab.org/add
- Official Docs: https://help.scilab.org/docs/6.1.1/en_US/atoms.html

Chapter 5

Xcos Architecture Analysis and Multi-Language Integration

5.1 The Background

This project was born out of a fundamental need within the open-source scientific community: to bridge the gap between two powerful yet disparate platforms, Octave and Scilab.

While Octave serves as a robust numerical computation environment, it lacks a native, userfriendly graphical modeling tool for complex systems.

This stands in stark contrast to commercial alternatives like Simulink and Scilab's own Xcos, which provide an intuitive visual environment for system simulation.

Recognizing this critical gap, the project was initiated with the ambitious goal of providing Octave users with a familiar visual tool for building and simulating complex systems. The initial hypothesis was that we could leverage the well-established architecture of Xcos to serve a new audience without a massive reengineering effort.

Our journey began by investigating the existing **sci_cosim** toolbox, a tool designed to facilitate communication between Scilab and other external software. The initial idea was to use this tool to transfer variables between Octave and Scilab and then simply initiate Xcos simulations remotely. This seemed like a straightforward solution, a way to connect two powerful worlds with minimal effort.

However, a deeper analysis of the sci_cosim source code revealed a more complex truth. It became clear that the toolbox was not a true integration tool but rather a sophisticated remote shell.

It did not facilitate native integration; instead, it would simply run the entire Xcos model on Scilab's platform while relaying information to Octave. This was inefficient, as it would require both Scilab and Octave to run in parallel, and it failed to achieve the seamless, platform-agnostic solution we were seeking.

This critical discovery shifted our entire approach, moving our focus from a simple remote-shell solution to a more profound investigation into Xcos's core architecture.

5.2 Xcos Architecture Analysis

Xcos is a graphical editor within the Scilab environment designed for the elegant modeling and simulation of hybrid dynamical systems. Its architecture is a sophisticated composition of three primary components that work in tandem to transform a visual model into executable simulation code.

5.2.1 Core Components

- **Xcos Editor**: This is the graphical user interface (GUI) for creating and editing Xcos diagrams, a component largely implemented in Java. It allows users to visually design systems by dragging and dropping blocks and connecting them to form a cohesive diagram.
- **Scicos Compiler**: This pivotal component translates Xcos diagrams and block definitions into simulation code. The Scicos team has clarified that while an early version of the compiler was a prototype written in the Scilab language, the most recent and significantly improved version is now written in OCaml. The compiler's role is to convert the user's high-level graphical model into a structured format that the Simulator can understand.
- **Simulator**: The true workhorse of the system, the Simulator is a substantial C program that executes the generated code to perform the simulation. It leverages highly optimized computational functions written in C, Fortran, or C++, which are independent of Scilab itself.

The Scicos team has clarified that Xcos serves as a graphical front-end for Scicos, and the core code for evaluation, compilation, and simulation resides within the scicos module. Block definitions are located in the scicos_blocks module. These folders also contain the C and FORTRAN routines that define the core computational functions for each block.

5.2.2 Simulation Workflow and Efficiency

Xcos employs a unique and highly efficient approach to simulation where the computational functions for various blocks are primarily written in C/FORTRAN and dynamically linked to the Xcos/Scilab environment.
This means that the actual simulation computations are not performed by Scilab directly, but rather by these external, pre-compiled C/FORTRAN functions.
This design choice is fundamental to the system's performance and is a key reason for its speed and efficiency.
Compared to interpreted languages like Scilab or Octave, C/FORTRAN is generally more suitable for computationally intensive tasks.
While it is possible to write computational functions in Scilab, this is mainly recommended for prototyping purposes. For production-level simulations, the use of dynamically linked C/FORTRAN functions is preferred due to their superior performance.

5.2.3 Interface and Computation Functions

For a block to be fully operational within Xcos, it relies on two essential functions: an Interface Function and a Computation Function.

- **Interface Function**: Also known as the Block Definition Function, this is a metadata-driven component responsible for defining the block's properties, setting its graphical representation, and specifying how it interacts with other elements in the Xcos environment. It associates the block with its corresponding computation function, but it does not perform any numerical computations itself. This function is not required if we can directly generate a valid .xcos file with the correct format.
- **Computation Function**: Also known as the Simulation Function, this is the core logic that processes input values, performs mathematical operations, handles state updates in dynamic systems, and manages real-time execution.

The execution flow during a simulation is a well-defined sequence of events. First, Xcos initializes the block using the interface function. Then, for each time step of the simulation, the simulation engine calls the computation function. This function processes its inputs, computes the outputs, and updates the system's state. This cycle continues until the simulation is complete.

5.3 Approaches for Integration

With the initial sci_cosim approach proving to be a dead end, our project shifted to a more direct and fundamental form of integration. Our investigation into the Xcos architecture revealed two potential pathways for linking Octave with the Scicos simulator, each with its own set of advantages and challenges.

5.3.1 Two Pathways to Integration

The first approach involved writing the computational functions directly in the Octave language. This would require integrating the Octave runtime with the Scicos simulator.
While this pathway seemed conceptually straightforward, it presented significant performance drawbacks. As our analysis confirmed, this would introduce an interpretive layer that would slow down simulations by a factor of 20 to 100 times compared to the native C/FORTRAN routines.
Given that one of Scilab's key selling points is its speed, this performance degradation would be unacceptable for many users.
The second and more promising approach was to use Octave's C++ API, liboctave, to write the computational functions. This method requires no extra integration, as C/C++ support is already a native part of the Scicos Simulator. By writing the core computational logic in C++, we could bypass the performance penalties of an interpretive language. The primary task would then be to port the interface and data structures to Octave to enable the generation of compatible .xcos files.

5.4 Challenges and Strategic Realizations

Our journey was not without its hurdles. One of the most significant challenges has been the scarcity of in-depth, low-level documentation for both Xcos and Scicos. This has made it difficult to fully grasp the system's underlying architecture and has required extensive source code analysis and direct consultation with the Scicos team. We also discovered that liboctave's documentation is somewhat deprecated and the APIs have changed over time, requiring us to stick with an older version of Octave (version 6 or older) to maintain stability.
This project, which began with the goal of a simple integration, has evolved into a grander vision. While we have confirmed that a simple Octave-based simulation would be inefficient from a performance standpoint, we have also realized that the real value lies in enhancing the user experience. Integrating Xcos with Octave would not be about performance gains but rather about providing Octave users with a familiar and user-friendly modeling tool, thereby enhancing accessibility and usability for the wider scientific community.

5.5 Conclusion and Future Direction

The initial objective of our project—to enable Octave users to leverage the powerful Xcos environment—remains central to our mission. However, our understanding of the problem has deepened, leading us to a more strategic and sustainable path forward.

- We have confirmed that the interface and data structures of Xcos can be ported to Octave, allowing us to generate valid .xcos files from within the Octave environment.
- We have decided to proceed with a new compiler and will use NSP as a key reference. The Scicos team member indicated that NSP is a more recent and improved version of Scicos with an OCaml-based compiler and a structure that is more Octave-like.
- The ultimate goal, aligned with Prof. Kanan's vision, is to make Xcos independent of Scilab and support its use from multiple languages. To achieve this, we plan to consult with Prof. Kumar and Prof. Kanan to decide between two strategic pathways: translating the complete system to Octave or designing a universal interface with core functionalities that can be accessed from any language. The latter, which we favor, requires a single major translation and only minor adjustments for each new language.

This journey has been one of continuous learning and adaptation. From an initial hypothesis that was quickly debunked, we have arrived at a clear, forward-thinking strategy that will not only meet our project's objectives but also contribute a valuable, open-source tool to the global community of engineers and researchers.

Chapter 6

Learnings

I have had a lot of great experiences from this internship opportunity, some are enumerated below.

1. Technical Skills Enhancement:
    - Gained proficiency in Scilab, Linux, dynamic linking, c++ Octave, Git, and GitHub.
    - Developed advanced coding skills like testing and documentation.
    - Improved understanding of technical concepts and practical applications.
2. Feedback and Continuous Improvement:
    - Learned to receive and act on constructive feedback.
    - Gained an understanding of the importance of continuous learning and improvement.
    - Developed the ability to self-assess and seek growth opportunities.
3. Professionalism and Work Ethic:
    - Developed a strong sense of professionalism in a workplace setting.
    - Learned the importance of punctuality, reliability, and accountability.
    - Gained experience in maintaining a professional demeanor in various sit– uations.
4. Adaptability and Flexibility:
    - Learned to adapt to new environments and changing circumstances.
    - Gained experience in managing multiple tasks and shifting priorities.
    - Developed resilience and the ability to thrive in a dynamic work environment- menu.

Chapter 7

Conclusion

My internship experience at FOSSEE, IIT Bombay, has been an enriching and intellectually stimulating journey. Having initially joined as a Semester-Long summer Intern in 2024, this Semester long Winter Internship (2024) has allowed me to further deepen my expertise and make meaningful contributions to the Scilab Signal Processing Toolbox.
Throughout this phase, I have focused on improving the functionality, efficiency, and usability of the toolbox while reinforcing my understanding of open-source development. A key aspect of my work involved enhancing and optimizing signal processing functions within Scilab. This included refining existing implementations, ensuring compatibility, and improving performance to reduce dependencies on external toolkits. Additionally, I contributed to resolving issues within the Scilab-Octave Toolbox, which I was not able to fix because of changes in the Scilab API. Also, the Scilab-Octave interface had various limitations and was inefficient in terms of memory and performance Beyond these contributions, I also conducted an architectural analysis of Xcos to explore its integration with programming languages and Octave. This investigation aimed to assess how Xcos can be leveraged for advanced simulations and computational workflows, potentially enhancing its interoperability with external tools and expanding its application scope. Looking ahead, I remain committed to continuing my contributions to FOSSEE projects, particularly in advancing the Signal Processing Toolbox. I am deeply grateful to my mentor, Ms. Rashmi Patankar, for her continuous guidance, encouragement, and support. Her insights have been instrumental in my growth throughout this internship. I also extend my sincere thanks to Prof. Kannan M. Moudgalya and Prof. Kumar Appaih for their valuable guidance, as well as my friends for their unwavering support. This internship has not only strengthened my technical and problem-solving skills but has also reinforced my passion for open-source development and scientific computing. I look forward to applying the knowledge and experience gained here to future projects and continuing my journey as an open-source contributor.

Signal processing Toolbox development

Reference

- https://github.com/abinash108/Signal-processing-toolkit-devlopment-
- https://github.com/FOSSEE/fossee-scilab-octave-toolbox
- https://octave.sourceforge.io/pkg-repository/signal/
- https://scilab.in/fossee-scilab-toolbox/signal-processing-toolbox
- http://www.scicos.org/
- Book : Modeling and Simulation in Scilab_Scicos with ScicosLab 4.4