# Module Interface Specification for FSL

Bo Cao

November 23, 2019

# 1 Revision History

| Date | Version | Notes |
|---|---|---|
| Nov. 26, 2019 | 0.99 | First Draft |

# 2 Symbols, Abbreviations and Acronyms

See SRS Documentation at https://github.com/caobo1994/FourierSeries/blob/master/docs/SRS/SRS.pdf. We also define the following acronyms for the scope of this document

| Acronym | Full Text |
|---|---|
| OOR | Out of range |

# Contents

# 3 Introduction

The following document details the Module Interface Specifications for Fourier Series Library (FSL).

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/caobo1994/FourierSeries/.

# 4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1|c_2 \Rightarrow r_2|...|c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by FSL.

| Data Type | Notation | Description |
|---|---|---|
| character | char | a single symbol or digit |
| integer | $\mathbb{Z}$ | a number without a fractional component in (-$\infty$, $\infty$) |
| natural number | $\mathbb{N}$ | a number without a fractional component in [1, $\infty$) |
| real | $\mathbb{R}$ | any number in (-$\infty$, $\infty$) |

The specification of FSL uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, FSL uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification. For simplicity, the sequence of type T will be abbreviated as seq(T), while that with dimensions $[l_1, ..., l_n]$ as seq(T, $l_1, ..., l_n$).

# 5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

| **Level 1** | **Level 2** |
|---|---|
| Hardware-Hiding Module | Infrastructure (hardware-related part, external) |
| Behaviour-Hiding Module | Data definition module<br>Conversion module<br>Transformation module<br>Basic operations module<br>Advanced operations module |
| Software Decision Module | Linear solver (external or partially external)<br>Integral (external or partially external)<br>Infrastructure (software-related part, external) |

Table 1: Module Hierarchy

# 6 MIS of Infrastructure

## 6.1 Module

Infrastructure.

## 6.2 Uses

None.

## 6.3 Syntax

## 6.4 Exported Data Type

FLOAT a floating point data type used in the library.

sequence the abstract sequence data type.

# 7 MIS of Data Definition

## 7.1 Module

Data Definition.

## 7.2 Uses

Infrastructure.

## 7.3 Syntax

### 7.3.1 Exported Constants

None.

### 7.3.2 Exported Data Type

There is one exported data type, `CFST`, which is the type of a CFS object. (abstract, with one type template `FLOAT`, indicating floating point types)

The structure of `CFST` is

- $n$: integer
- $\omega$: `FLOAT`
- $A$: seq(`FLOAT`)

- $B$: seq(`FLOAT`)

The mathematical notation is

$$\text{CFST} := \text{tuple of } (n : \mathbb{N}, \omega : \text{FLOAT}, A : \text{TA}, B : \text{TB}) \tag{1}$$

where TA and TB are

$$\begin{aligned} \text{TA} :=& seq(\texttt{FLOAT}, n) \\ \text{TB} :=& seq(\texttt{FLOAT}, n+1) \end{aligned} \tag{2}$$

During implementation, it is recommended that the sequence type used above are random accessible.

### 7.3.3 Exported Access Programs

The exported access programs for this module are mainly getters and setters. For simplicity, we use the following getter and setter rules.

For each variable X, the getters and setters are

| Name | In | Out | Exceptions |
|---|---|---|---|
| `getX` | | `X`: type of X | None. |
| `setX` | `X`: type of X | | None. |

Furthermore, we design getters and setters for each element of $A$ and $B$. For each X being A or B, the syntax is

| Name | In | Out | Exceptions |
|---|---|---|---|
| `getXi` | | `X`: type of X | OOR. |
| `setXi` | `X`: type of X | | OOR. |

## 7.4 Semantics

### 7.4.1 Assumptions

### 7.4.2 Access Routine Semantics

This part follows the most common rules of getters and setters. For simplicity, we do not elaborate on them.

# 8 MIS of Linear Solver

## 8.1 Module

Linear Solver.

## 8.2 Uses

Infrastructure.

## 8.3 Syntax

### 8.3.1 Export Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| LinSolve | m: $\mathbb{N}$<br>A: seq(FLOAT, m, m)<br>b: seq(FLOAT, m) | x: seq(FLOAT, m) | Solution non-exist<br>Solution not unique |

## 8.4 Semantics

### 8.4.1 Assumptions

### 8.4.2 Access Routine Semantics

This part follows the most common semantics of linear solvers, and for simplicity, we do not elaborate on it.

# 9 MIS of Integral

## 9.1 Module

Integral.

## 9.2 Uses

Infrastructure.

## 9.3 Syntax

### 9.3.1 Export Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| Integral | f: FLOAT→FLOAT<br>a, b: FLOAT | res: FLOAT | Integral non-exist<br>or not computable |

## 9.4 Semantics

### 9.4.1 Access Routine Semantics

`Integral(f, a, b)`:

- output: $\int_a^b f(t)\mathrm{d}t$
- exception: Evident

# 10 MIS of Conversion

## 10.1 Module

Conversion.

## 10.2 Uses

Data Definition.

## 10.3 Syntax

### 10.3.1 Export Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| `ConvertFrom` | $n \in \mathbb{N}$<br>$\omega$: `FLOAT`<br>$A$: sequence of `FLOAT`<br>$B$: sequence of `FLOAT` | `CFST CFS` | MISMATCH_SIZE: Mismatch |
| `ConvertTo` | `CFST CFS`<br>user-acquired space for outputs | Same as inputs of `ConvertFrom` | None |

## 10.4 Semantics

## 10.5 Access Routine Semantics

With `M` being short for `MISMATCHED_SIZE`, ConvertFrom$(n, \omega, A, B)$:

- output: CFS.$n$, CFS.$\omega$, CFS.$A$, CFS.$B := n, \omega, A, B$
- exception: exc $:= (|A| \neq n + 1 \Rightarrow M || B| \neq n \Rightarrow M)$

The semantics for `ConvertTo` is straightforward, and we do not elaborate on it.

# 11 MIS for Transformation

## 11.1 Module

Transformation.

## 11.2 Uses

Data Definition, Integration.

## 11.3 Syntax

### 11.3.1 Export Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| `TransformTo` | $f \in \{\mathbb{R} \to \mathbb{R}\}$<br>$n \in \mathbb{N}$<br>$\omega \in \mathbb{R}^+$ | `CFST CFS` | None. |
| `FunctionValue` | `CFST CFS`<br>$t \in \mathbb{R}$ | $V \in \mathbb{R}$ | None. |

## 11.4 Semantics

## 11.5 Access Program Semantics

`TransformTo`$(f, n, \omega)$:

- output: $\text{CFS}.n, \text{CFS}.\omega, \text{CFS}.A, \text{CFS}.B := n, \omega, A, B$, where $A_i$ and $B_i$ are computed in accordance with T1 of the SRS.

`FunctionValue`$(\text{CFS}, t)$:

- output: $V := \sum_{i=0}^{\text{CFS}.n} \text{CFS}.A(i)\cos(i\omega t) + \sum_{i=1}^{\text{CFS}.n} \text{CFS}.B(i)\sin(i\omega t)$

| Name | In | Out | Exceptions |
|---|---|---|---|
| `ConvertFrom` | $n$: integer<br>$\omega$: `FLOAT`<br>$A$: sequence of `FLOAT`<br>$B$: sequence of `FLOAT` | Constructed CFS | Mismatch between $n$, (size of $A$ -1), and size of $B$ |
| `ConvertTo` | CFS<br>pointers to the space for outputs | Same as inputs of `ConvertFrom` | None |
| `TransformTo` | function $f$, $n$, $\omega$ | Constructed CFS | Any exception raised by integral module |
| `FunctionValue` | CFS, $t$ | Value of $f(t)$, where CFS comes from $f$ | None. |
| `Addition` | `CFST CFS1, CFST CFS2` | `CFST, CFSres` | mismatch of $n$ and $\omega$ |
| `Subtraction` | `CFST CFS1, CFST CFS2` | `CFST, CFSres` | Same as `Addition` |
| `Multplication` | `CFST CFS1, CFST CFS2` | `CFST, CFSres` | Same as `Addition` |
| `Divison` | `CFST CFS1, CFST CFS2` | `CFST, CFSres` | Same as `Addition`, plus any exception raised by linear solver and reclassified in `Division`. |
| `ToleratedEquality` | `CFST CFS1, CFST CFS2, FLOAT tol` | `Bool res` | Same as `Addition` |
| `Function` | `CFST CFS`, Taylor series type variable | `CFST, CFSres` | Insufficient length of Taylor series. |
| `Amplitude` | CFS | `FLOAT amp` | None. |

## 11.6 Semantics

### 11.6.1 State Variables

[Not all modules will have state variables. State variables give the module a memory. —SS]

### 11.6.2 Environment Variables

[This section is not necessary for all modules. Its purpose is to capture when the module has external interaction with the environment, such as for a device driver, screen interface, keyboard, file, etc. —SS]

### 11.6.3 Assumptions

[Try to minimize assumptions and anticipate programmer errors via exceptions, but for practical purposes assumptions are sometimes appropriate. —SS]

### 11.6.4 Access Routine Semantics

[accessProg —SS]():

- transition: [if appropriate —SS]
- output: [if appropriate —SS]
- exception: [if appropriate —SS]

[A module without environment variables or state variables is unlikely to have a state transition. In this case a state transition can only occur if the module is changing the state of another module. —SS]

[Modules rarely have both a transition and an output. In most cases you will have one or the other. —SS]

### 11.6.5 Local Functions

[As appropriate —SS] [These functions are for the purpose of specification. They are not necessarily something that is going to be implemented explicitly. Even if they are implemented, they are not exported; they only have local scope. —SS]

# References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

# 12 Appendix

[Extra information if required —SS]