Module Interface Specification for FSL

Bo Cao

December 10, 2019

[The multilined table looks funny when I use makecell to make a cell with multiple lines. I would be thankful if someone could enlighten me with some better ways. —Author]

1 Revision History

Date	Version	Notes
Nov. 26, 2019	0.99	First Draft
Dec. 10, 2019	1.0	Final version

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
MC	Mismatched CFS

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 (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)

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`, $l_1, ..., l_n$). Furthermore, we use both `Var: Type` and `Type Var` to indicate that variable `Var` is of type `Type`.

For consistency with mathematical notations, we denote $A : \mathbb{B}$ as $A \in \mathbb{B}$, when $\mathbb{B}$ is the notation of a common mathematical set, and A is a member of $\mathbb{B}$.

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 Conversion module Transformation module Basic operations module Advanced operations module
Software Decision Module	Linear solver (external or partially external) Integral (external or partially 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
- ω: `FLOAT`
- A: seq(`FLOAT`, n)

- B: seq(`FLOAT`, $n+1$)

The mathematical notation is

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

where TA and TB are

$$\text{TA} := \text{seq}(\texttt{FLOAT}, n)$$
$$\text{TB} := \text{seq}(\texttt{FLOAT}, n+1)$$

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 of the function intended for the element X_i is

Name	In	Out	Exceptions
`getXi`	$i \in \mathbb{N}$	`V: FLOAT`	OOR.
`setXi`	$i \in \mathbb{N}$, `V: FLOAT`		OOR.

For convenience, we may use `A(i)` and `B(i)` to represent `getAi(i)` and `getBi(i)`, respectively.

Based on the setters, we implement a constructor for `CFST`.

Name	In	Out	Exceptions
`new CFST`	n, ω, A, B: same type as their namesake in `CFST`	`CFST CFS`	None.

7.4 Semantics

[Rather than putting the details as an exported type, you should have state variables here to store the state information. You have behaviour that is more involved than simply using a tuple. There are different types of getters and setters, and you have exceptions. You also need a constructor so you create objects of this type. —SS]

[I have thought of exported type versus state variables, and I think exported variables for some reason. My understanding is that people use state variables when the program mainly involves the transition of states (members of this type) under given conditions, much like a (finite) state machine. As for exported data type, I tend to think that this is for a set of members (loosely) combined as an set of variables. —Author]

7.4.1 Assumptions

7.4.2 Access Routine Semantics

We give a template for `getX`, `setX`, `getXi` and `setXi` respectively. `setX(Xin)`:

- transition: $X = Xin$.

`getX()`:

- output: X.

`setXi(i, V)`:

- transition: $X[i - (offset)] = V$
- exception: exc := $((i < offset) \vee (i > n) \Rightarrow \text{OOR})$

`getXi(i)`:

- output: $X[i - offset]$
- exception: exc := $((i < offset) \vee (i > n) \Rightarrow \text{OOR})$

In these expressions, *offset* is 0 for `X=A` and 1 for `X=B`.

[Your shortened notation is fine, but sometimes it is less work for the reader if you just expand your definitions. —SS][Elaborated. A lot of stuff is too basic if the reader meets the requirement is the SRS, so I choose not to say that. I guess that I might put too much requirements in the SRS, but I think that users do not know what they want to get, and what they are getting if they do not meet the requirements. —Author]

`new CFST(n, ω, A, B)`

- transition: $CFS.n := n$, $CFS.\omega := \omega$, $CFS.A := A$, $CFS.B := B$
- output: CFS

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`: $\mathbb{N}$ A: seq(`FLOAT`, `m`, `m`) b: seq(`FLOAT`, `m`)	x: seq(`FLOAT`, `m`)	Solution non-exist Solution not unique

8.4 Semantics

8.4.1 Assumptions

8.4.2 Access Routine Semantics

`LinSolve(m, A, b)`:

- output: x that suits $Ax = b$.
- exception: exc := (Solution non-exist$\vee$Solution not unique $\Rightarrow$ Cannot return solution)

[This would be better if you elaborated on it. Our goal is to be unambiguous. It wouldn't be that much work to specify what a linear solver returns. —SS][Done. —Author]

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` `a`, `b`: `FLOAT`	res: `FLOAT`	Integral non-exist 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: Integral non-exist $\vee$ Integral not computable $\Rightarrow$ Integral not found [The programmer may not know as much as you. This isn't enough information on the possible exceptions. —SS][Done. —Author]

10 MIS of Conversion

10.1 Module

Conversion.

10.2 Uses

Data Definition.

10.3 Syntax

10.3.1 Export Access Programs

[We say $n : \mathbb{N}$ —SS][I know, but I am used to this way. I have made a notation about this in Section 4, Notation. —Author][Move comments here from the table, to have table not corrupted by the comments. —Author]

Name	In	Out	Exceptions
ConvertFrom	$n \in \mathbb{N}$ ω: FLOAT A: sequence of FLOAT B: sequence of FLOAT	CFST CFS	OOR: $\omega \leq 0$ MC: Mismatch between n, (size of A -1), and size of B
ConvertTo	CFST CFS	Same as inputs of ConvertFrom	None

10.4 Semantics

10.5 Access Routine Semantics

ConvertFrom(n, ω, A, B):

- output: *new* $CFST(n, \omega, A, B)$ [You want to call your constructor here —SS][Reorganized, forgot to implement a constructor. Guess I have written C for quite sometime. —Author]
- exception: exc := ($\omega \leq 0 \Rightarrow$ OOR$||A| \neq n + 1 \Rightarrow$ MC$||B| \neq n \Rightarrow$ MC)

ConvertFrom(CFS):

- output: $n := CFS.n, \omega := CFS.\omega, A := CFS.A, B := CFS.B$

[You might find that others do not find all of your steps as straightfoward as you do. —SS][Done here. —Author]

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`	$f \in \{\mathbb{R} \to \mathbb{R}\}$ $n \in \mathbb{N}$ $\omega \in \mathbb{R}^+$	`CFST CFS`	None.
`FunctionValue`	`CFST CFS` $t \in \mathbb{R}$	$V \in \mathbb{R}$	None.

11.4 Semantics

11.5 Access Program Semantics

`TransformTo`(f, n, ω):

- output:

$$
\begin{aligned}
A_0 :=& (1/2\pi) \int_{-\pi}^{\pi} f(t)\mathrm{d}t, \\
A_i :=& (1/\pi) \int_{-\pi}^{\pi} f(t)\cos(i\omega t)\mathrm{d}t, \\
B_i :=& (1/2\pi) \int_{-\pi}^{\pi} f(t)\sin(i\omega t)\mathrm{d}t \\
A :=& < A_0, ..., A_n >, \\
B :=& < B_1, ..., B_n >, \\
\mathrm{CFS}.n, \mathrm{CFS}.\omega, \mathrm{CFS}.A, \mathrm{CFS}.B :=& n, \omega, A, B
\end{aligned}
$$

`FunctionValue`(CFS, t):

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

12 MIS of Basic Operation

12.1 Module

Basic Operations.

12.2 Uses

Data Definition, Linear Solver

12.3 Syntax

12.3.1 Export Access Programs

Name	In	Out	Exceptions
CFSMatch	CFST CFS1, CFST CFS2	Bool res	None.
Addition	CFST CFS1, CFST CFS2	CFST CFSres	MC
Subtraction	CFST CFS1, CFST CFS2	CFST CFSres	MC
Multplication	CFST CFS1, CFST CFS2	CFST CFSres	MC
Divison	CFST CFS1, CFST CFS2	CFST CFSres	MC
Amplitude	CFST CFS1	FLOAT amp	None.

12.4 Semantics

12.4.1 Access Program Semantics

CFSMatch(CFST CFS1, CFST CFS2):

- output: $(\text{CFS1}.n = \text{CFS2}.n) \wedge (\text{CFS1}.\omega = \text{CFS2}.\omega) \Rightarrow \text{TRUE}|\text{TRUE} \Rightarrow \text{FALSE}$

The semantics of Addition, Subtraction, and Multiplication are similar in structure, and the only difference is the calculation of the A and B variables shown below, which is consistent with the corresponding theories introduced in SRS. That is, the implementer shall choose A and B in IM4 of SRS for Subtraction, IM5 for Multiplication, IM6 for Division. [It is a good idea to reference the SRS. It would be better if your reference were specific to models (chunks) in the SRS. The less work the reader has to do the better. You want the translation from MIS to code to be almost mechanical. You also want it to be possible for someone that is less knowledgeable than you on the topic of your software. Ideally, a second year student in CS or SE should be able to implement your modules. —SS][Added here accordingly. BTW, I never hope that a second year student in CS/SE of McMaster can implement this module. I know that their mathematics skill sucks, far, far away from implementing this module. —Author]

As an example, we give the semantics of the Addition function.

Addition(CFST CFS1, CFST CFS2):

- output: $A_i := \text{CFS1.getAi}(i) + \text{CFS2.getAi}(i)$, $B_i := \text{CFS1.getBi}(i) + \text{CFS2.getBi}(i)$,
  $A :=< A_0, ..., A_{\text{CFS1.n}} >$, $B :=< B_0, ..., B_{\text{CFS1.n}} >$,
  $\text{CFSres}.n, \text{CFSres}.\omega, \text{CFSres}.A, \text{CFSres}.B := \text{CFS1}.n, \text{CFS1}.\omega, A, B$
- exception: $\text{exc} := (\text{CFSMatch}(\text{CFS1}, \text{CFS2}) = \text{FALSE} \Rightarrow \text{MC})$

As for `Division`, the difference is much significant. The A and B is computed as follows

$$\begin{aligned} x :=& \text{LinSolve}(2n+1, M, y), \\ A :=& x[0, \text{CFS1}.n+1], \\ B :=& x[\text{CFS1}.n+1, |x|] \end{aligned}$$

where M and y are constructed in accordance with the theories for division in SRS.
`Amplitude(CFST CFS1)`:

- output: $amp := \sqrt{\text{CFS1.getAi}(0)^2 + (1/2) * \sum_{i=1}^{\text{CFS1}.n}(\text{CFS1.getAi}(i)^2 + \text{CFS1.getBi}(i)^2)}$

13 MIS for Advanced Operation

13.1 Module

Advanced Operation.

13.2 Uses

Basic Operation.

13.3 Syntax

In the following section, `TST` is a function type $\mathbb{Z}^* \to$ FLOAT, and for any object `TS` of this type associated with a mathematical function, `TS(i)` gives the `i`-th Taylor coefficient of this mathematical function.

13.3.1 Export Access Programs

Name	In	Out	Exceptions
`ToleratedEquality`	`CFST CFS1, CFST CFS2, FLOAT tol`	`Bool res`	MC
`Power`	`CFST CFS`, $m \in Z^*$	`CFST CFSres`	None.
`Function`	`CFST CFS, TST TS`	`CFST CFSres`	None.

13.4 Semantics

13.4.1 Access Routine Semantics

`ToleratedEquality(CFST CFS1, CFST CFS2, FLOAT tol)`:

- output: res := (Amplitude(Subtraction(CFS1, CFS2)) $\leq$ tol $\Rightarrow$ TRUE|TRUE $\Rightarrow$ FALSE)

- exception: exc := (CFSMatch(CFS1, CFS2) = FALSE $\Rightarrow$ MC)

`Power(CFST CFS,` $m \in Z^*$`):`

- output:

$$A :=< 1, 0, ..., 0 >, |A| = (n+1);$$
$$B :=< 0, 0, ..., 0 >, |B| = n$$
$$\text{CFSzero}.n, \text{CFSzero}.\omega, \text{CFSzero}.A, \text{CFSzero}.B := n, \omega, A, B$$
$$\text{CFSres} := (m = 0 \Rightarrow \text{CFSzero}|\text{TRUE} \Rightarrow \text{Multiplication}(\text{CFS}, \text{Power}(\text{CFS}, m-1)))$$

`Function(CFST CFS, TST TS):`

- output: CFSres := $\sum_{i=0}^{\text{CFS}.n}(1/i!)\text{TS}(i)\text{Power}(\text{CFS}, i)$

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.

[Extra information if required —SS] [Content added here. I expect to add some guidelines for implementations, but I forgot to mention here, since at the time of first draft, I did not consider about implementation too much. —Author]

A Guild Line for Implementing in C++

This module can be implemented in C++ by doing a direct translation from mathematics notation to C++. The following translation rules are recommended and expected to be followed.

- `seq(A)` and `seq(A, m)` shall be implemented as `std::vector<A>`, while `seq(A, m, n)` shall be implemented as `std::vector<std::vector<A>>`. The size of each `std::vector` shall be checked manually, and exceptions shall be thrown if needed.
- Type $\mathbb{N}$ shall be implemented as `size_t`. This type is available after `#include <vector>;`.
- Type $\mathbb{R}$ shall be implemented as `FLOAT`, while type $\mathbb{R} \to \mathbb{R}$ shall be implemented as `std::function<FLOAT(FLOAT)>`.