1 User manual for Scilab2C

This section describes steps to be followed for using 'Scilab2C'. Pre-requisites are mentioned followed by procedure to install 'Scilab2C'.

1.1 Installation

1.1.1 Prerequisites

There are few prerequisites or some packages must be pre installed before we can use 'Scilab2C'. These are:

- Scilab >=5.5.1.
- Scilab-Arduino toolbox (If using 'Scilab2C' to generate code for Arduino)
- Arduino makefile (https://github.com/sudar/Arduino-Makefile). Install using 'sudo apt-get install arduino-mk'.
- BCM2835 C library for RasberryPi (http://wiringpi.com/)
- RasberryPi tools (For cross compiling code for RasberryPi)

  Detailed instructions for installing these packages are in section 'Installing supporting packages'

1.1.2 Installing Scilab2C

Before we can use 'Scilab2C' extension, we need to install latest version of Scilab2C. Follow following procedure to get latest source code from github repo.

- Open terminal window. (Ctrl+Shift+T is shortcut).
- Change current directory to '/path/to/scilab/share/scilab/contrib'. Normally it is in '/usr/share/' if installed using system interface. Replace '/path/to/' by actual path to folder 'scilab'. For example, if you have installed Scilab using system interface ('apt-get' on Ubuntu), then run following command in terminal:
  `cd /usr/share/scilab/share/scilab/contrib`
- Clone the git repo using following command:
  `git clone https://github.com/siddhu8990/Scilab2C.git`
- Make sure a directory named 'Scilab2C' is present in 'contrib' folder.

- Open the Scilab.
- Run 'builder.sce file present in 'Scilab2C/2.3-1 using 'exec'. This generates binary files from source files.
  `exec('/path/to/Scilab2C/2.3-1/builder.sce')`
- In 'Home/.Scilab/scilabx.x.x' make a new file '.scilab' if it does not exist already. (You may need to enable 'Show hidden files' from 'View' menu to see .Scilab folder'. Open '.scilab using suitable editor. Add following line in this file:
  `exec('/path/to/Scilab2C/2.3-1/loader.sce')`
  This will load the 'scilab2c' everytime scilab is started.

1.1.3 Installing supporting packages

Most of the supporting packages or libraries which are required are provided with the toolbox. But they were compiled using latest source code available at release of toolbox. If you want to use latest libraries, steps to compile the same are listed below. You can follow these steps and replace old files with newly generated ones.

- **'scilab-arduino toolbox'** Latest version of 'scilab-arduino toolbox is available through 'Atoms', toolbox installer module for Scilab.
- **RasberryPi tools**
  - Make a folder named 'RasberryPi tools' somewhere on the harddisk.
  - Open terminal and change directory to 'RasberryPi tools'. Clone 'Tools' repo using 'git clone https://github.com/raspberrypi/tools.git'.
  - Add location of toolchain to your 'PATH' variable.
    `'export PATH=$PATH:/location/of/tools/folder/arm-bcm2708/gcc-linaro-arm-li`
- **WiringPi C library for RaspberryPi**
  - Download latest source code from '`https://git.drogon.net/?p=wiringPi`'. Extract source files at some suitable location.
  - Copy these source files to RaspberryPi at suitable location. Follow instructions given for installation.
  - **not complete**
- **Cross compiling Lapack and Blas for RasberryPi**
  - Download latest source code for Lapack from '`http://www.netlib.org/lapack/`'. Extract source files at some suitable location.

- Open file 'make.inc.example' given in Lapack folder using some editor.
- Edit following items as shown:
  * FORTRAN = arm-linux-gnueabihf-gfortran
  * LOADER = arm-linux-gnueabihf-gfortran
  * CC = arm-linux-gnueabihf-gcc
  * ARCH = arm-linux-gnueabihf-ar
  * RANLIB = arm-linux-gnueabihf-ranlib
- Since we are cross compiling for some other platform, normal way compiling will not work.
- Open terminal window and change current directory to laplack directory.
- We will need to compile BLAS, CBLAS and Lapack separately and in same order.
- Change current directory to /path/to/lapack/BLAS/SRC and run 'make'. This will generate 'librefblas.a' in Lapack folder.
- Now change current directory to /path/to/laplack/CBLAS and run 'make'. This will generate 'libcblas.a' in Lapack folder.
- Now change current directory to /path/to/laplack/SRC and run 'make'. This will generate 'liblapack.a' in Lapack folder. Now replace the generated lib files in 'src/c/hardware/rasberrypi/libraries' in 'scilab2c' source folder.

- **GNU Scientific Library (GSL) for RasberryPi**
  - Before going further, make sure that you have installed 'RasberryPi tool' following the instructions given here.
  - Get latest source code for GSL from `ftp://ftp.gnu.org/gnu/gsl/`.
  - Extract source code at some suitable location on harddrive.
  - Open the terminal window and change current directory to the location where source is extracted.
  - Execute following command
    `./configure -host=arm CC=arm-linux-gnueabihf-gcc ar=arm-linux-gnueabihf-ar --enable-static`
  - Then execute 'make libgsl.la' to cross compile the library. Don't do 'make install' as it is noramlly next step.
  - Library 'libgsl.a' is created in folder '.libs'. By default this folder is hidden.

– Now replace the generated lib file in 'src/c/hardware/rasberrypi/libraries' in 'scilab2c' source folder.

– We need to set a environment variable 'C_INCLUDE_PATH' so that arm compiler can find required files while compiling the code. For doing this, type following in the terminal:
`export C_INLUDE_PATH="/path/to/gsl2.1/folder"`
Replace /path/to/gsl2.1/folder by actual path on your machine.

1.2 Using Scilab2C for C code generation

Scilab2C extension in Scilab can be used for generating C code from a Scilab script. Currently it supports four target platforms:

- **Standalone C code**: General C code which can be compiled using any compiler
- **Arduino** : Arduino sketches can be generated using Scilab scripts written using 'Scilab-Arduino toolbox' (A scilab-arduino extension is required)
- **AVR** : C code can be generated for using hardware peripherals of AVR microcontroller
- **Raspberry Pi** : C code for using hardware peripherals of Raspberry Pi can be generated

You can follow following steps for generating C code using Scilab2c extension for required target platforms.

1.2.1 Generating standalone C code

1. Write the Scilab script first which is to be converted to C. Scilab code can contain single file or many files, but each file must be a Scilab function. There must be one main Scilab file in case project contains many files, from which execution of code starts. All Scilab files must be in a single folder.
2. Before a Scilab file can be translated to a c code, some function annotations should be added manually. Function annotations gives information about no. of inputs/outputs, their types etc. Refer 'Function annotations' for more details.
3. Type 'sci2c_gui' or 'scilab2c' in scilab console. This will prompt the GUI of Scilab2C toolbox as shown in figure. 1

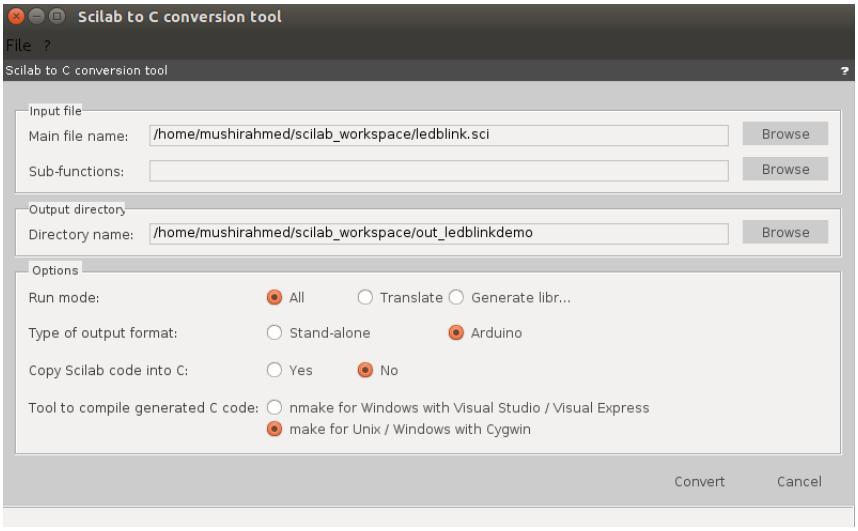

Figure 1: GUI for 'Scilab2C'

4. Click 'Browse' next to 'Main file name' textbox, browse to location of main scilab file and select it. (Refer figure 2)

5. If Scilab code contains many files, select folder containing these file by clicking 'Browse' next to 'Sub-functions' textbox.

6. Create a new folder somewhere on the disk, preferably in same folder containing Scilab files. Select this newly created folder by clicking 'Browse' next to 'Directory name' textbox. (Refer figure 3). Generated C code files are stored in this folder.

7. Choose appropriate options from 'Options' box. Different options are explained below:

   (a) Run mode : If only directory structure is to be generated in output directory, select 'Generate library'. If only conversion of scilab files is to be done, select 'Translate'. In case both are to be done, select 'All'.

   (b) Target platform : To generate standalone C code, select 'Standalone C' from dropdown. (Refer figure 4)

   (c) Copy Scilab code into C: Select 'Yes' or 'No' accordingly.

   (d) Tool to compile generated C code: Select appropriate option depending upon platform on which generated code will be complied.

8. Confirm everything again and then press 'Convert' button. (Refer figure 5)

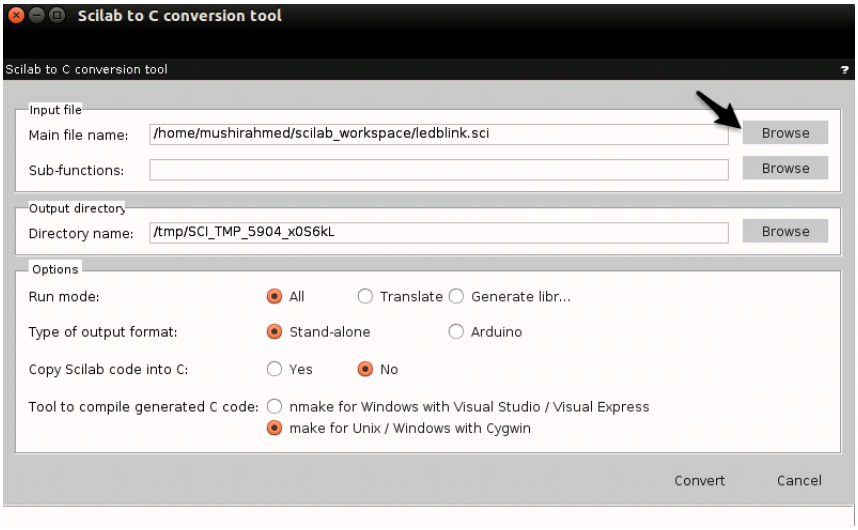

Figure 2: Select 'main' scilab file for conversion

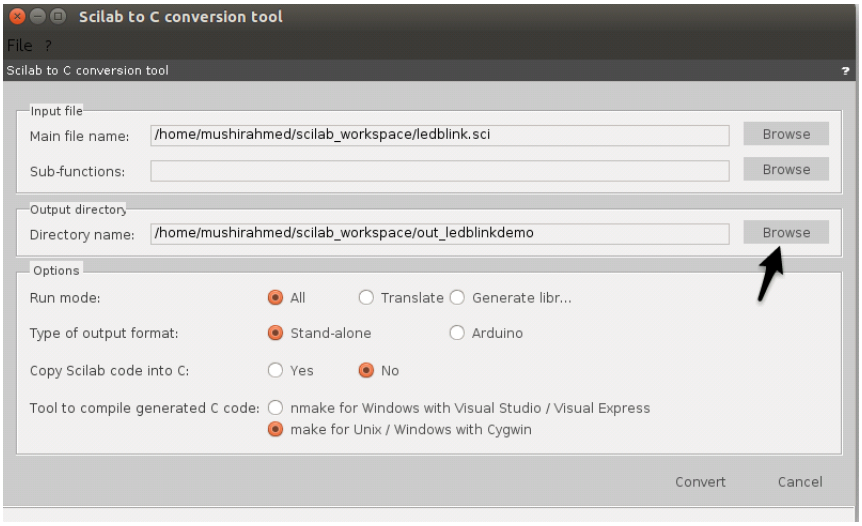

Figure 3: Select output folder

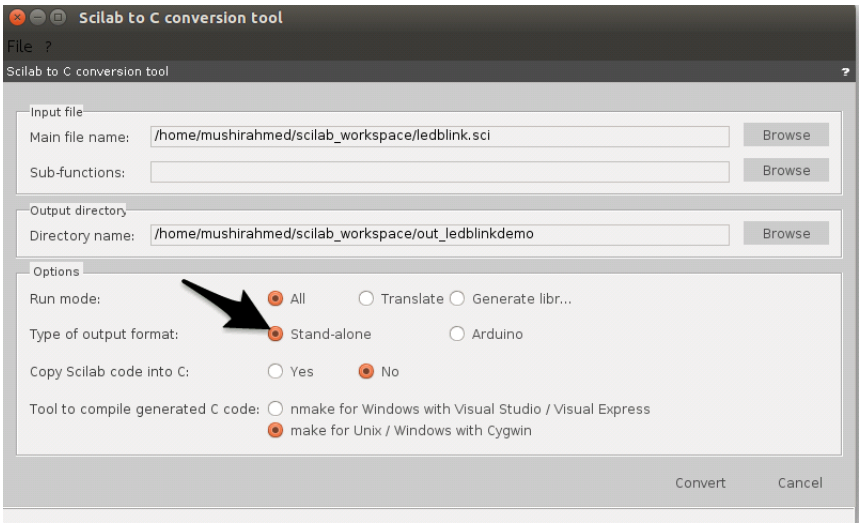

Figure 4: Select 'Standalone C' from dropdown

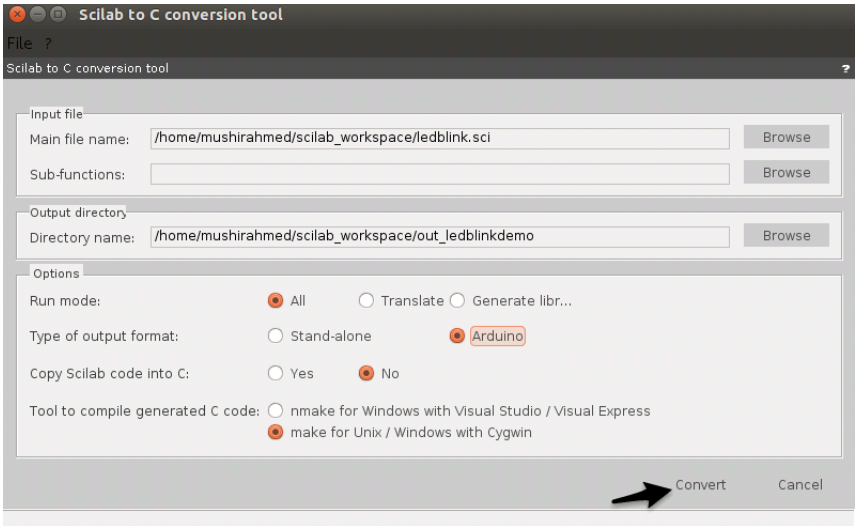

Figure 5: Select output folder

9. After clicking 'Convert', Scilab code will be run in Scilab, to check for any errors. If code runs successfully, a prompt will occur asking if you want to continue to code conversion or not. Select 'Yes'. If Scilab code doesnt run correctly then code conversion is stopped there itself. Correct the Scilab code and follow the steps again.

10. After selecting 'Yes' for code conversion, code conversion starts. If code conversion is done successfully, you will see the message in command window.

11. Generated code can be seen in output folder. By default a makefile is generated which uses 'GCC' compiler to compile the C code. You can compile this code using 'make'. Open output folder in terminal and type 'make' and press Enter. Once code is compiled successfully, it is run in terminal and output can be seen in terminal window. Check the output for correctness. If code did not behave as expected, correct the Scilab code and follow the process again.

1.2.2 Generating code for Arduino

1. Write the Scilab script first which is to be converted to C. Scilab code can contain single file or many files, but each file must be a Scilab function. There must be one main Scilab file in case project contains many files, from which execution of code starts. All scilab files must be in a single folder. You can verify working of Scilab script by runnig it on an Arduino board. Modify the script untill code behaves as expected. Once script is finalised, remove the commands 'open_serial' and 'close_serial'.

2. Type 'sci2c_gui' or 'scilab2c' in scilab console. This will prompt the GUI of Scilab2C toolbox as shown in figure 1

3. Click 'Browse' next to 'Main file name' textbox, browse to location of main scilab file and select it. (Refer figure 2)

4. If scilab code contains many files, select folder containing these file by clicking 'Browse' next to 'Sub-functions' textbox.

5. Create a new folder somewhere on the disk, preferably in same folder containing scilab files. Select this newly created folder by clicking 'Browse' next to 'Directory name' textbox. (Refer figure 3)

6. Choose appropriate options from 'Options' box. Different options are explained below:

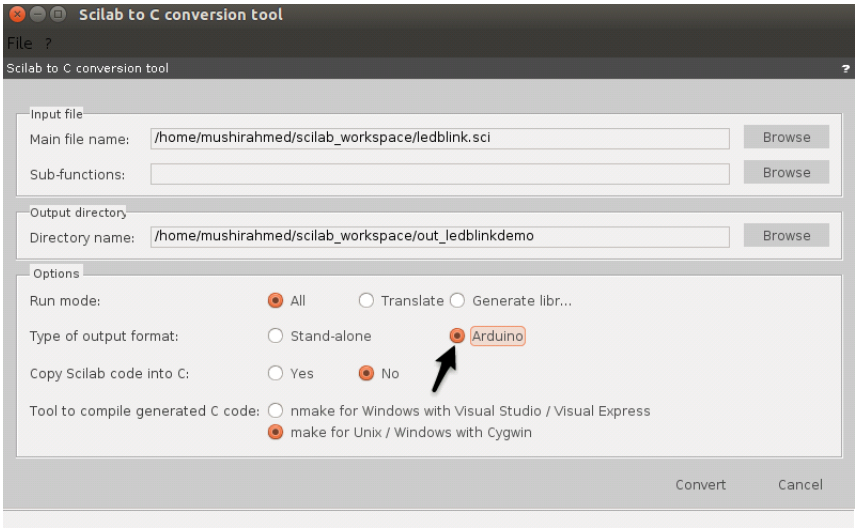

Figure 6: Select 'Standalone C' from dropdown

(a) Run mode : If only directory structure is to generated in output directory, select 'Generate library'. If only conversion of scilab files is to be done, select 'Translate'. In case both are to be done, select 'All'.

(b) Target platform : To generate C code for arduino, select 'Arduino' from dropdown. (Refer figure 6)

(c) Copy scilab code into C: Select 'Yes' or 'No' accordingly.

(d) Tool to complie generated C code: Select appropriate option depending upon platform on which generated code will be complied.

7. Confirm everything again and then press 'Convert button. (Refer figure 5)

8. Code conversion will start, promting different messages in command window. If conversion completes successfully, prompt will occur in command window indicationg the same.

9. Generated code can be seen in output folder. A separate folder named 'Arduino' is created, which contains a makefile and an arduino sketch file – sci2c_arduino.ino.

10. Open 'Makefile' using suitable text editor. Change following parameters according to board and connection:

    (a) BOARD_TAG

    (b) ARDUINO_PORT

11. Open the terminal and change current directory to the directory containing modified Arduino sketch and then compile by typing 'make' in terminal.

12. If code is compiled successfully, you can upload it to arduino using 'make upload' command.

13. If code doesnot behave as expected, modify Scilab code and follow the steps again.

1.2.3 Generating code for AVR

1.2.4 Generating code for Raspberry Pi

1.3 Function Annotations

Each scilab function/file should start with the function annotation section having following structure:

```
//SCI2C: NIN=
//SCI2C: NOUT=
//SCI2C: OUT(1).TP=
//SCI2C: OUT(1).SZ(1)=
//SCI2C: OUT(1).SZ(2)=
//SCI2C: OUT(2).TP=
//SCI2C: OUT(2).SZ(1)=
//SCI2C: OUT(2).SZ(2)=
...
//SCI2C: OUT(NOUT).TP=
//SCI2C: OUT(NOUT).SZ(1)=
//SCI2C: OUT(NOUT).SZ(2)=
//SCI2C: DEFAULT_PRECISION= DOUBLE
```

Although a minimum flexibility is available in the function annotation, we suggest observing anyway the following annotation rules:

- Each annotation line must start with //SCI2C: tag. This makes possible to hide annotations to Scilab interpreter and makes also possible to run the code without error generation.

- The first line of the Scilab file to be translated must start with the number of input arguments annotation //SCI2C: NIN=.

- The number of output annotations must be equal to NOUT.

- No blank lines should be inserted in the annotation section.

- The = symbol used in the assignment cannot be separated from the annotation specifier:
  - The following annotation is correct: //SCI2C: OUT(2).TP= ...
  - The following annotation is wrong: //SCI2C: OUT(2).TP = ...
- To be sure that the annotation of the user code has been correctly interpreted by Sci2C please check the .ann file generated by Sci2C when the .sci file is read. Supposing that we are translating file myfun.sci, the user should access the myfun.ann file generated by Sci2C in order to check that it contains the right annotations.

Each of the above tag is explained below:

1. **NIN**
   NIN specifies the number of input arguments that the function can handle. This tag is useful for Sci2C functions that can handle different number of input arguments, whereas it is not useful for User2C functions because they must work with a fixed number of input arguments. NIN annotation tag makes use of the following syntax:

   //SCI2C: NIN= number

   where:
   number is a number specifying the number of input arguments.

2. **NOUT**
   NOUT specifies the number of output arguments the function can handle. This tag is useful for Sci2C functions that can handle different number of output arguments, whereas it is not useful for User2C functions because they must work with a fixed number of output arguments. NOUT annotation tag makes use of the following syntax:

   //SCI2C: NOUT= number

   where:
   number is a number specifying the number of output arguments.

3. **TP**
   This tag specifies the type (and precision) of the returned output arguments. The annotation section must contain a number of TP annotation tags equal to the number of output arguments. TP annotation tag makes use of the following syntax:

//SCI2C: OUT(k).TP= type expression

where:
k is a sequential number (from 1 to NOUT) indicating that we are annotating the type and precision of the k-th output argument.
type expression is an expression that specifies the type and precision of the k-th output argument. Type expression can be a composition of the type annotation functions listed below. In the following list, for each type annotation function it is specified its number of input and output arguments, and the result returned:

- **FA_TP_S**: NInArgs = 0, NOutArgs=1; when this function is invoked it means that the output argument is of s type (real, float single precision).
- **FA_TP_D**: NInArgs = 0, NOutArgs=1; when this function is invoked it means that the output argument is of d type (real, float double precision)..
- **FA_TP_C**: NInArgs = 0, NOutArgs=1; when this function is invoked it means that the output argument is of c type (complex, float single precision).
- **FA_TP_Z**: NInArgs = 0, NOutArgs=1; when this function is invoked it means that the output argument is of z type (complex, float double precision).
- **FA_TP_UINT8**: NInArgs = 0, NOutArgs=1; when this function is invoked it means that the output argument is of u8 type (unsigned, 8 bit precision).
- **FA_TP_UINT16**: NInArgs = 0, NOutArgs=1; when this function is invoked it means that the output argument is of u16 type (unsigned, 16 bit precision).
- **FA_TP_INT8**: NInArgs = 0, NOutArgs=1; when this function is invoked it means that the output argument is of i8 type (signed, 8 bit precision).
- **FA_TP_INT16**: NInArgs = 0, NOutArgs=1; when this function is invoked it means that the output argument is of i16 type (signed, 16 bit precision).
- **FA_TP_USER**: NInArgs = 0 NOutArgs=1; when this function is invoked it means that the output argument must be specified by the user in the Scilab code. More specifically, the type and precision can be

specified in the Scilab code by using the following data annotation functions: float, double, floatcomplex, doublecomplex (see section dedicated to data annotation for more details).

- **IN(m).TP**: NInArgs = 0 NOutArgs=1; when this function is invoked it will return the type and precision of the m-th input argument.

4. **SZ**
   This tag specifies the size of the returned output arguments. The annotation section must contain a number of SZ annotation tags equal to twice the number of output arguments, this is because for each output argument two SZ annotations are required, the first one specifying the number of rows and the second one specifying the number of columns of the output argument. SZ annotation tag makes use of the following syntax:

   //SCI2C: OUT(k).SZ(1)= size expression
   //SCI2C: OUT(k).SZ(2)= size expression

   where: k is a sequential number (from 1 to NOUT) indicating that we are annotating the size of the k-th output argument. *.SZ* is assumed to be a 2-element string array indicating the number of rows (.SZ(1)) and columns (.SZ(2)) of the k-th output argument. Number of rows and columns can be specified by using numbers or symbols. *size expression* is an expression that specifies the size of the k-th output argument. Size expression can be a composition of the size annotation functions listed below. For each size annotation function it is specified its number of input and output arguments, and the result returned:

   - **FA_SZ_1**: NInArgs = 1, NOutArgs=1; this function extracts the first element of a two-element string array. It is useful to extract the number of rows from the size of an input argument as shows the following example:

     //SCI2C: OUT(k).SZ(2)= FA_SZ_1(IN(m).SZ)

     In this annotation we are indicating that the number of columns (.SZ(2)) of the k-th output argument is equal to the number of rows of the m-th input argument. An equivalent annotation is the following one:

     //SCI2C: OUT(k).SZ(2)= IN(m).SZ(1)

   - **FA_SZ_2**: NInArgs = 1, NOutArgs=1; this function extracts the second element of a two-element string array. It is useful to extract the number of columns from the size of an input argument as shows the following example:

//SCI2C: OUT(k).SZ(1)= FA_SZ_2(IN(m).SZ)

In this annotation we are indicating that the number of rows (.SZ(1)) of the k-th output argument is equal to the number of columns of the m-th input argument. An equivalent annotation is the following one:

//SCI2C: OUT(k).SZ(1)= IN(m).SZ(2)

- **FA_SZ_OPDOTSTAR**: NInArgs = 2, NOutArgs=1; this function accepts two input .SZ string arrays and returns a .SZ string array which specifies the size of the output argument returned by the .∗ operator. This is a useful function to annotate functions that work with two input arguments and return a single output argument whose size is a function of the sizes of the input arguments according to the rules used for the .∗ operator. For example for "./", ".∗" "." operators the following size annotations can be adopted:

//SCI2C: OUT(1).SZ(1)=
FA_SZ_1(FA_SZ_OPDOTSTAR(IN(1).SZ,IN(2).SZ))
//SCI2C: OUT(1).SZ(2)=
FA_SZ_2(FA_SZ_OPDOTSTAR(IN(1).SZ,IN(2).SZ))

- **FA_SZ_OPHAT**: NInArgs = 2, NOutArgs=1; this function is an alias for FA_SZ_OPDOTSTAR. This is because ˆ and .∗ operators have the same behaviour for what concerns the size of the output argument.
- **FA_SZ_OPMINUS**: NInArgs = 2, NOutArgs=1; this function is an alias for FA_SZ_OPDOTSTAR. This is because "−" and ".∗" operators have the same behaviour for what concerns the size of the output argument.
- **FA_SZ_OPPLUSA**: NInArgs = 2, NOutArgs=1; this function is an alias for FA_SZ_OPDOTSTAR. This is because "+" and ".∗" operators have the same behaviour for what concerns the size of the output argument.
- **FA_SZ_OPSTAR**: NInArgs = 2, NOutArgs=1; this function accepts two input .SZ string arrays and returns a .SZ string array which specifies the size of the output argument returned by the ∗ operator. This is a useful function to annotate functions that work with two input arguments and return a single output argument whose size is a function of the sizes of the input arguments according to the rules used for the ∗ operator. See the following example:

//SCI2C: OUT(1).SZ(1)=
FA_SZ_1(FA_SZ_OPSTAR(IN(1).SZ,IN(2).SZ))

//SCI2C: OUT(1).SZ(2)=
FA_SZ_2(FA_SZ_OPSTAR(IN(1).SZ,IN(2).SZ))

- **FA_ADD**: NInArgs = 2, NOutArgs=1; this function accepts two input strings returns a string which contains the sum of the two input strings according to the following rules:

  FA_ADD('3','43') = '46'
  FA_ADD('symbol1','43') = 'symbol1+43'
  FA_ADD('symbol1','symbol2') = 'symbol1+symbol2'

  As shown in the examples above FA_ADD performs a sum of the two input strings when both strings contain numbers, otherwise the output string will be a composition of the two input strings with the "+" symbol. This function is used to annotate functions that generate outputs whose size is given by adding the sizes of the input arguments. Let's consider, as example, the OpRc operator which implements the row concatenation ("[,]"). Row concatenation is shown in the following example:
  A = [1 2 3; 3 4 5];
  B = [4 5; 1 1];
  C = [A,B]
  According to the code above C is equal to [1 2 3 4 5; 3 4 5 1 1]
  In terms of size, the number of rows of C is equal to the number of rows of A (or B) and the number of columns of C is equal to the number of columns of A plus the number of columns of B. It follows that the right annotation for the OpRc operator is:

  //SCI2C: NIN= 2 //SCI2C: NOUT= 1 //SCI2C: OUT(1).TP=
  FA_TP_MAX(IN(1).TP,IN(2).TP) //SCI2C: OUT(1).SZ(1)=
  IN(1).SZ(1) //SCI2C: OUT(1).SZ(2)=
  FA_ADD(IN(1).SZ(2),IN(2).SZ(2))

- **FA_SUB**: NInArgs = 2, NOutArgs=1; this function has the same behaviour of FA_ADD, but it performs a subtraction between the two input arguments.
- **FA_MUL**: NInArgs = 2, NOutArgs=1; this function has the same behaviour of FA_ADD, but it performs a multiplication between the two input arguments.
- **FA_DIV**: NInArgs = 2, NOutArgs=1; this function has the same behaviour of FA_ADD, but it performs a division between the two input arguments.

- **FA_MAX**: NInArgs = 2, NOutArgs=1; this function has the same behaviour of FA_ADD, but computes the maximum between the two arguments. When the two input arguments don't specify a number, the output argument will be equal to the first input argument. See the following examples:
  FA_MAX('3','55') = '55'
  FA_MAX('3','a') = '3'
  FA_MAX('cccc','a') = 'cccc'
  FA_MAX('cccc','88888888888888') = 'cccc'

- **FA_INT**: NInArgs = 1, NOutArgs=1; this function truncates to int the input argument only if the input argument is a string specifying a number. See the following examples:
  FA_INT('3.444') = '3'
  FA_INT('-3.444') = '-3'
  FA_INT('ciao') = 'ciao'