USER MANUAL

Version 2.6.0

MATLAB®

CONTENTS

OVERVIEW

What is eVariX™?

The eVariX™ technology relies on a source-to-source compiler, Cold©, which transforms MATLAB® codes into optimized C++11 codes for sequential or parallel execution on CPU or GPU, and relying on a specific library, called Cold© Run-Time library (CRT©). In addition, the eVariX™ package provides the `evarix` driver which directly produces stand-alone executables or MEX files and relieves you from dealing with the back-end C++ compiler options.

What's new?

eVariX™ 2.6.0 features several new capabilities, among which:

- partial support of MATLAB® classes (Section 4.6);
- support of the MATLAB® `load` function (Section 4.8.4 and 7);
- calling functions from external libraries directly from the MATLAB® code (Chapter 3);
- automatic GPU code generation for code portions delimited by annotations (Section 4.8.3) .

Document Organization

Part I focuses on how to use eVariX™. The installation of the eVariX™ package on Windows and Linux systems is described in Chapter 1, while Chapter 2 extensively presents the `evarix` command-line interface and its options. Chapter 3 describes how to call functions from an external library.

The eVariX™ compiler deliberately supports a subset of the MATLAB® language in order to generate efficient code. This means that the input code must follow a set of rules to be eVariX™*-compliant*. These rules are described and explained in Chapter 4. This chapter also describes the *annotation*[1] language provided by eVariX™ to drive the compilation or optimization of the code. Supported functions are listed in Part II.

Licences and disclaimers of tools used by and/or distributed with eVariX™ are given in Chapter 9.

Numerical precision

The codes generated by eVariX™ don't rely on the same libraries as the initial MATLAB® input models when executed in their native environment. This, as well as the back-end C++ compiler options, may have an impact on the numerical results, which may slightly differ from those obtained in the native MATLAB® environment. In most cases, the differences are not significant[2]. However, should it be so, you may first try to reduce the optimization level of the back-end C++ compiler (see Section 2.6), and turn off all eVariX™ optional optimizations. If the issue persists, please contact us so that we can identify the source of the problem.

[1] These annotations take the form of *pragmas* and both terms are interchangeably used in this document.

[2] We have sometimes observed differences in the order of 10^{-15}.

Release contents

- eVarix™ compiler (Cold©): `cold-2.6.4`
- eVarix™ driver: `evarix-1.7.3`
- Cold© run-time library: `crt-2.6.3`

Support

Any question, remark or bug can be reported by email at:

evarix.support@silkan.net

Part I

Using eVariX™

CHAPTER 1

INSTALLATION

This chapter describes how to install eVariX™ on Windows systems (see Section 1.1) and on Linux systems (see Section 1.2). Please note that this eVariX™ release requires a C++11 compiler.

1.1 Windows

1.1.1 Prerequisites

C++-language compiler: MinGW-w64 distribution

The MinGW-w64 compiler is supported[1]. Please make sure that the MinGW-w64 distribution is fully C++-11 compliant (G++-4.8 or higher).

Note

All the tools provided by the MinGW-w64 package, in particular g++, must be accessible via the PATH environment variable. Please make sure to add the bin/ directory to the variable. You can indifferently use the Windows console or the console provided by MinGW.
To check your MinGW installation, open a terminal and type:

```
g++ --version
```

1.1.2 Installing eVariX™

1. Launch the `evarix-2.6.0-win_x64.exe` installer and follow the instructions. The default installation directory is:

   C:\Program Files\SILKAN\evarix-2.6.0

2. Add to the PATH environment variable the directory

   C:\Program Files\SILKAN\evarix-2.6.0\bin

Setting environment variables on Windows systems

On Windows systems, the environment variables can be set in

Computer → System Properties → Advanced System settings → Environment variables.

3. Run the `rlmhostid` program in

   C:\Program Files\SILKAN\evarix-2.6.0\etc

[1] This compiler is available at http://mingw-w64.org/doku.php.

and send the generated hostID to your eVariX™ provider, who will send you a license file.

4. Copy this license file to

C:\Program Files\SILKAN\evarix-2.6.0\etc

1.1.3 Checking the installation

Using your favorite text editor, create the following test file, called `sample.m`[2]:

```
A = [1 2 3 4 5];
B = [5 4 3 2 1];
A * B'
A'* B
```

Then, in a terminal, type:

```
evarix sample.m -o sample.exe
sample
```

The program must display:

```
ans =

    35


ans =

     5     4     3     2     1
    10     8     6     4     2
    15    12     9     6     3
    20    16    12     8     4
    25    20    15    10     5
```

If the compiler cannot be found, check the PATH environment variable value (see Section 1.1.1).

Note

During the compilation with MinGW, the following warnings may be issued several times, without any effect on the correctness of the process:

```
Warning: .drectve `-defaultlib:"uuid.lib" ' unrecognized
Warning: corrupt .drectve at end of def file
```

[2] This file can be found in `C:\Program Files\SILKAN\evarix-2.6.0\samples\matlab`.

1.2 Linux

1.2.1 Prerequisites

C++-language compiler

eVariX™ requires a C++11 compiler, either `g++` (GNU, version 4.8 or higher) or `icc` (Intel, version 14.0 or higher).

1.2.2 Installing eVariX™

1. Extract the `evarix-2.6.0-linux_x64.tar.gz` file at the desired location.
2. For your convenience, add the `EVARIX_INSTALL_DIR/bin` directory to the `PATH` environment variable.

Setting environment variables on Linux systems

On Linux systems, the environment variables can be set with:

- `export VAR=VALUE` in `sh` shells;
- `setenv VAR VALUE` in `*csh` shells.

3. Run the `rlmhostid` program located in `EVARIX_INSTALL_DIR/etc` and send the generated hostID to your eVariX™ provider, who will send you a license file.
4. Copy this licence file to `EVARIX_INSTALL_DIR/etc`.

1.2.3 Checking the installation

Using your favorite text editor, create the following test file, called `sample.m`[3]:

```
A = [1 2 3 4 5];
B = [5 4 3 2 1];
A * B'
A'* B
```

Then, in a terminal, type:

```
evarix sample.m -o sample
./sample
```

The program must display:

```
ans =

    35


ans =

     5     4     3     2     1
    10     8     6     4     2
    15    12     9     6     3
    20    16    12     8     4
    25    20    15    10     5
```

[3]This file can be found in `EVARIX_INSTALL_DIR`/samples/matlab

1.2.4 Graphics library

eVariX™ offers a minimal support for graphics functions, see imread and imshow. These functions require the OpenCV library[4], with a version higher than 3.3. You can install it from official Linux distribution repositories, or from binaries or source codes from the OpenCV website. In the latter case, you may need to modify eVariX™ configuration file (see 2.6) to set the path in which the OpenCV libraries are installed. To locate them, type in your terminal:

```
locate libopencv_core.so
```

The result should look like something similar to:

```
/usr/local/lib/libopencv_core.so
/usr/local/lib/libopencv_core.so.3.3
/usr/local/lib/libopencv_core.so.3.3.1
```

Then, in the configuration file, modify the line beginning with

```
$OPENCV_PATH =
```

by adding the full path to `libopencv_core.so`. In the previous case:

```
$OPENCV_PATH = /usr/local/lib
```

[4] http://opencv.org/

CHAPTER 2
COMPILING

eVariX™ core component is Cold©, a MATLAB®-to-C++ compiler. It takes `.m` files as input and produces a corresponding C++-code file. The latter can then be compiled to generate either a stand-alone executable from a script file (and possibly some function files); or a binary MEX file from a function.

This document presents the `evarix` driver, a tool managing the whole compilation process, by successively calling Cold© and the required back-end compiler to get an executable or a MEX file from a MATLAB® file.

The first section of this chapter describes how to invoke `evarix` in command-line mode to generate a stand-alone executable from a script. The MEX generation process is then detailed in Section 2.2. Section 2.3 describes how to use eVariX™ directly from the MATLAB® environment (Linux only). `evarix` and Cold© options are finally presented in Sections 2.4 and 2.5, and the configuration of `evarix` in Section 2.6.

2.1 Compiling a script with eVariX™ command-line tool

The syntax for calling `evarix` is:

```
evarix inFile [-o outFile] [Options]
```

where `inFile` is the input MATLAB® file (or C++ file) and `outFile` is the name of the binary file generated by the process. For example, the `sample.m` file can be compiled into an executable named `sample` with the command:

```
evarix sample.m -o sample
```

By default, Cold© looks for additional function files in the current directory, but other paths can be specified with the appropriate option (see Section 2.5).

Calling `evarix` on a MATLAB® code chains the following steps:

1. MATLAB® to C++ compilation *via* Cold©;
2. Compilation of the C++ generated file using the back-end compiler;
3. Link edition with Cold© run-time libraries.

Calling `evarix` on a C++ file previously generated by Cold© (with the `--cold-only` or `--keep` options) only performs the last two steps:

```
evarix sample.cold.cpp -o sample
```

2.2 Generating a MEXfile

EVARIX™ can generate a MEX file from a MATLAB® function or script. This allows to use the optimized binary code directly from the MATLAB® environment. It has been tested on Linux systems only.

This feature requires to specify the entry point (namely the function name) and the type of the input arguments (the types of the output arguments are inferred by EVARIX™). It can be done either with the `--pragma-entry` option (see Section 2.5.5), or with the `entry` annotation (see Section 4.8.4). The syntax is presented in Section 4.2.5.

Let us consider the following code:

```
function C = myAddition(A, B)
    C = A + B;
end
```

Without any information about the types of the variables `A` and `B`, EVARIX™ can neither infer the type of `C` (see section 4.2), nor generate a typed C++ function prototype. The `entry` annotation can be used to specify these types. For example, to get a C++ function returning the sum of two matrices of doubles:

```
%pragma cold entry(myAddition, matrix{double}, matrix{double})
function C = myAddition(A, B)
    C = A + B;
end
```

Once the MATLAB® code has been annotated, it can be compiled using the `--mex` option. `evarix` calls COLD© with the required options and then back-end compiler.

As an example, producing a MEX file for the function `myAddition` is done with the command:

```
evarix --mex myAddition.m -o myDoubleMatrixAddition
```

The name of the compiled function in MATLAB® is then `myDoubleMatrixAddition`.
Note: using this option, the value of the `--output` option must be the name of the new compiled function being added to MATLAB®, and not the name of a file.

To use the Intel ICC compiler instead of g++, type:

```
evarix --mex -cc=icc myAddition.m -o myDoubleMatrixAddition
```

The MEX file can then be used as a built-in MATLAB® function:

```
>> X = [1 2; 3 4]
>> Y = [5 6; 7 8]
>> Z = myDoubleMatrixAddition(X,Y)
```

To avoid conflicts with libraries used by MATLAB®, the latter must be lauched with a pre-loading of the libraries provided with EVARIX™:

```
export LIBCUSTOMMKL=$EVARIX_INSTALL_DIR/lib/2.6.3/mkl/libcustomMKL.so
export LIBIOMP5=$EVARIX_INSTALL_DIR/lib/2.6.3/mkl/libiomp5.so

LD_PRELOAD=$LIBCUSTOMMKL:$LIBIOMP5 matlab
```

2.3 Using eVariX™ from the MATLAB® Environment (Linux only)

The eVariX™ package provides a set of functions which can be used in the MATLAB® environment for compiling MATLAB® codes and running the generated codes. To import these functions in the MATLAB® environment, the `EVARIX_INSTALL_DIR/bin/` path must be added to the environment from the console:

```
>> addpath('EVARIX_INSTALL_DIR/bin/');
```

Three functions are provided and described below:

- `evarix_exec` to compile a script or function call into an executable and execute it immediately ;
- `evarix_compile` to compile a script or function into a loadable MEX binary file ;
- `evarix_get_entry_format` to get the representation of the type/shape of a variable in the eVariX™ syntax for an entry option or annotation.

⚠ Limitation

The compiled script or function must be *self-contained* and musn't have any effect on the MATLAB® environment:

- Global or MATLAB® workspace variables cannot be imported from the calling environment, and if the script/function creates (global) variables, they won't be known in the calling environment after returning from the call. A workaround is to pass variables as parameters to the function to compile.
- In the compiled script/function, calls to the MATLAB® functions are replaced with calls to the Cold© runtime library functions, which have no impact on the MATLAB® environment. For instance, calls to the `rand` function won't have any effect on the MATLAB® random number generator, which may be an issue if it is invoked before or after the call to the compiled script/function.

Demos The `samples` directory of eVariX™ installation contains a demo script `evarix_in_matlab_demo.m`. All the usages of the `evarix_exec` and `evarix_compile` functions are illustrated.

```
function [argsout] = evarix_exec(funcname, varargin)
```

This function replaces the MATLAB® `run` function: it compiles the input script or function by invoking `evarix` with the provided input options, and runs the generated executable, with the provided input arguments in case of a function. If the compilation process fails, the `run` function is invoked so that you actually get the expected result.

There are several possible usages:

- `evarix_exec('script')`
  compiles and executes the input *script* or function with no parameter.
  Note: it is possible to pass either `'script'` or `'script.m'`, but in both case, the directory containing the file must be in the MATLAB® search path (use `addpath` function if needed).
- `evarix_exec('script', {'opt1', .. , 'optk'})`
  compiles the input *script* with the `opt1, .. ,optk` options, and then executes the generated binary.
- `evarix_exec('func', param1, .. , paramk)`
  `[outs] = evarix_exec('func', param1, .. , paramk)`
  compiles the input *function* with no option, and then executes the generated binary with the given input parameters `param1, .. , paramk`. If the function has outputs, they can be collected as usual.

- `evarix_exec('func', {'opt1', .. , 'optk'}, param1, .. , paramk)`
  `[outs] = evarix_exec('func', {'opt1', .. , 'optk'}, param1, .., paramk)`
  compiles the input *function* with the `opt1,..,optk` options, and then executes the generated binary with the given input parameters `param1, .., paramk`.

Examples of usage of `evarix_exec`

- Compiling and excuting a simple script (`script_demo2_pp.m`):

```
% script_demo2_pp.m
A=reshape(1:100, 4, 25)
B=reshape(1:100, 4, 25)
disp('C=A+B');
%pragma cold parallel elementwise
C=A+B;
disp(C);
```

  - with no option:

```
evarix_exec('script_demo2_pp');
```

  - with parallel annotations enabled and a stack-size of 10MB (see `--stack-size`):

```
evarix_exec('script_demo2_pp', ...
            {'--enable-parallel-pragmas', '--stack-size=10MB'});
```

- Compiling and executing a function with two parameters and two outputs:

```
function [C,D]=func_demo5_2out(A,B)
%pragma cold parallel elementwise
C=A+B;
D=A*B;
end
```

  - with no compilation option:

```
T=rand(20);
U=rand(20);
[C,D]=evarix_exec('func_demo5_2out', T, U);
```

  - with parallel annotations enabled:

```
T=rand(20);
U=rand(20);
[C,D]=evarix_exec('func_demo5_2out','--enable-parallel-pragmas',T,U);
```

```
function [status, options, vars] = evarix_compile(filename, funcname, varargin)
```

The input parameters are:

- *filename* is the MATLAB® file containing the function or script,
- *funcname* is the name of the compiled function to generate
- *varargin* contains a cell array of options, (*{opt1,...,optk}*) to pass on to `evarix`, and variables (*var1, ..., vark*) of the same types/shapes as the input parameters of the function . They are used to give to `evarix` the types and shapes of the parameters that will be later passed on to the compiled function, and are thus mandatory. However, their values are not considered during the compilation process.

In most cases, the output parameters can be ignored. They are:

- *status*, which is true if the compilation succeeds, false otherwise;
- *options* is a cell array of strings containing the complete list of options passed on to `evarix` (namely *{opt1,...,optk}*);
- *vars* is a cell array with the variables passed on to the `evarix_compile` function to account for the types/shapes of the function input parameters (`{var1, ..., vark}`).

Possible usages are:

- `evarix_compile('script', 'script_lib')`
  compiles the input script into a MEX file.
- `evarix_compile('script', 'script_lib', {'opt1', .. ,'optk'})`
  compiles the input script with the *opt1, .. ,optk* options, and generates a MEX file.
- `evarix_compile('func', 'func_lib', var1, .. , vark)`
  compiles the input function with no entry annotation from the input file, and generates a MEX file.
- `evarix_compile('func', 'func_lib', [])`
  compiles the input function taking no argument and with no entry annotation, and generates a MEX file. The empty brackets account for the empty list of parameters to be passed to the function.
- `evarix_compile('func', 'func_lib')`
  compiles the input function with an entry annotation in the input file, and generates a MEX file.
- `evarix_compile('func', 'func_lib', {'opt1', .. , 'optk'}, var1, .. , vark)`
  compiles the input function with no entry annotation and with the `{'opt1', .. ,'optk'}` options, and generates a MEX file.

Once a function or script has been compiled, it is added to the current workspace and can be used as a usual MATLAB® user function, such as in the following example:

```
A = magic(10);
B = magic(10);

evarix_compile('func_demo1_add.m', 'bar', A, B);
bar(A,B);

[st, opt, vars] = evarix_compile('func_demo1_add.m', 'bar2', n, A);
if (st)
   bar2(vars{:})
end
```

Reusing MEX file

Both processes generate a MEX file: `script.mexa64` or `funcname.mexa64`. When MATLAB® is restarted, the MEX file can be invoked again by adding the path to its location, without loading the EVARIX™ functions in the environment:

```
addpath('/path/to/bar/mex/file');
A = magic(20);
B = magic(20);
bar(A,B);
```

```
function entry_fmt = evarix_get_entry_format(varsargs)
```

varargs is list of parameters of a supported type and *entry_fmt* is the representation of the types/shapes of the input parameters in the EVARIX™ syntax for an entry option or annotation.

```
>> evarix_get_entry_format([1 2; 3 4], 'foo', 5)

ans =

matrix{int},string,int
```

2.3.1 Linux, the standard C++ library, and the MKL libraries

On linux systems, depending on the version of MATLAB® you use, there can be conflicts between the standard C++ library used by MATLAB® and the one used by the executables compiled by EVARIX™. A workaround is to preload the C++ library used by the C++ compiler called by `evarix` when launching MATLAB®:

```
LD_PRELOAD=/the/complete/path/to/your/libstdc++.so  matlab
```

With `g++` and `icc`, you can get the path to the standard C++ library with:

```
g++ -print-file-name=libstdc++.so
```

Similarly, there can be conflicts between the MKL library version used by MATLAB® and the one used by eVariX™. In addition, if the code generated by eVariX™ uses OpenMP constructs, directly or indirectly, and depending on the C++ compiler you use, there may be conflicts between the OpenMP library used by eVariX™ and the OpenMP library used by MATLAB®. A workaround is also to preload the custom eVariX MKL library and its OpenMP library. If `MKL_LIB_DIR` is defined as `EVARIX_INSTALL_DIR/lib/2.6.3/mkl/`, then the command is:

```
LD_PRELOAD=$MKL_LIB_DIR/libcustomMKL.so:$MKL_LIB_DIR/libiomp5.so matlab
```

2.4 eVariX™ driver Options

This section presents the options of `evarix`, with short options names when available. Any option not recognized by `evarix` is passed on to Cold©.

2.4.1 General Options

--cold-only

Perform Cold© compilation only and solely produce a C++ code file.

--help or -h

Display usage and list all options.

--keep

By default, the C++ source file produced by Cold© is deleted at the end of the `evarix` process. The `--keep` option allows to keep this file.

--output <filename> or -o <filename>

Set the name of:

- the output file in the case of compilation of a standalone application: name of the C++ file with the `--cold-only` option, name of the executable otherwise;
- the new MATLAB® command if the `--mex` option is used (see section 2.2).

--verbose or -v

Increase verbose information level during `evarix` process.

--version

Display the `evarix` driver version information.

--advanced-options

Display the list of `evarix` and Cold© advanced options (experienced users).

2.4.2 Backend C++ Compiler Options

-cc=<CppCompiler>

Use `CppCcompiler` instead of G++. If `<CppCompiler>` is found in the configuration file (see Section 2.6), the corresponding options are applied.

--Wc=<option>

Pass `option` on to the back-end compiler.

--Wl=<option>

Pass `option` on to the linker.

--Wm=<option>

Pass `option` on to the MEX compiler.

2.4.3 Advanced Options

--configuration-file=<configFile>

Select a configuration file (see Section 2.6).

--select-configuration=<configName>

Select a specific configuration in the default or selected configuration file (see Section 2.6).

--check-install-matlab

Check installation for Matlab configuration.

--check-install-mex

Check installation for MEX configuration.

--check-install

Check default installation.

--check-overall

Check installation of all compilers and libraries.

OpenCV

The `OpenCV` library can be reported as missing using the check* options, even if present on the machine. This is not an issue if the code does not use `imread` or `imshow` functions, or if the library is in the default search paths.

2.5 eVariX™ Compiler Options

This section presents the options of the eVariX™ compiler, Cold©, with short options names when available.

2.5.1 General Options

--help, -h

Display usage and list all options.

--path=<directory>

If your application is defined in several files, this option allows to set `directory` as the root of a path in which COLD© will recursively search for functions files. Multiple path roots may be provided using several `--path` options.

By default, the current working directory is added to the search path, but is not searched recursively for function files. For that purpose you must explicitly use "`--path=.`".

Note that conventionally, a file defining a function *must* bear the name of the function followed by the extension `.m`, otherwise COLD© will not be able to find the function.

⚠ Limitation

eVariX™ does not support scripts calling other scripts.

--verbose, -v

Increase verbose level.

--version

Display the version of the COLD© compiler.

--complex

The type of the output of some mathematical functions (log, pow, sqrt...) can be different depending on the chosen codomain. This option forces the computation of these functions to be done in $\mathbb{C}$.

By default, for such functions, the computation is done in $\mathbb{R}$ for performance reasons. Undefined values, for example `sqrt(-1)`, are set to `nan`.

--advanced-options

Display the list of advanced options (experienced users).

2.5.2 Code generation & back-end

--import-lib=/path/to/libmylib.m

Specifies the path to an external library description file (see Chapter 3).

--mex

Call the MathWorks MEX compiler (see 2.2) to produce a MEX binary file callable in the MATLAB® environment.

--no-comment, -nc

Do not print comments of MATLAB® code in the generated C++ code.

--no-prefix, -np

By default, all user variable names are prefixed in the generated C++ code, to avoid potential name clashes with other variable names generated by COLD©. With this option, user defined variable are not prefixed in the generated C++ code (not recommended).

--output=<filename>, -o=<filename>

Set the name of the output file.

--stack-size=<size>[B|kB|mB|gB]

The executables generated by EVARIX™ use an internal stack for storing some of the variables used by the program. The default stack size is set to 1 giga-bytes. The `--stack-size` option allows to change this size to `size` bytes if no unit is provided, or to `size[B|kB|mB|gB]` otherwise, `B` standing for "bytes". `size` must be an integer.

It is difficult to evaluate the amount of stack memory required for an application since the stack stores the user-defined variables and the temporary variable generated by COLD©.

If the allocated memory is not large enough, the execution of the generated code will stop with an error when trying to use more memory than allocated. On the contrary, allocating too much memory can degrade performances (mainly because of swapping).

2.5.3 Optimization options

--inline-level=<level>

Inline expansion is a code transformation that replaces calls to functions by their code. The integer parameter can take three different values:

- `--inline-level=0` (default) turns off inline expansion.
- `--inline-level=1` expands calls to functions specified by the user, either through options --inline-function or --inline-pattern, or through inlining pragmas.
- `--inline-level=2` expands calls to functions specified by the user, and uses a heuristics to also expan calls to other user functions.

Inline expansion has the advantage of exposing more information to the compiler and can in most cases enhance the performance of the generated code. However, it may also have a negative impact on the size of the application, which, in extreme cases, may slow down the compilation as well as the generated code. This is why complementary options (`--inline-function` or `--inline-function-pattern`) and inlining annotations must be used with care.

Functions with several `return` statements and recursive functions are currently not inlined, as well as functions returning several values and called from within a complex expression.

Note

This option and the associated complementary options are in beta version.

--inline-function=<name>

When inline expansion is turned on (inline level is 1 or 2), specifies the name of a user function which must be expanded inline, regardless of COLD© internal heuristics. A warning is issued if the function can't be inlined.

--inline-function-pattern=<string>

When inline expansion is turned on (inline level is 1 or 2), all the user functions whose name begins with `<string>` are expanded inline, regardless of COLD© internal heuristics. A warning is issued if the functions can't be inlined.

Example:

```
evarix --inline-level=1 --inline-function-pattern=bar script.m -o script
```

expanses all functions whose name begins with `bar`.

--inline-add-comments

Surround inlined code by comments giving the name of the inlined function and the actual/formal parameter correspondances.

--fes-level=<level>

When enabled, Forward Expression Substitution (FES) propagates constant values and expressions (either scalar or multi-dimensional) to their usages when it is profitable to ease other program transformations or for performance reasons. The integer parameter can take three different values:

- `--fes-level=0` (default) turns off FES.
- `--fes-level=1` turns on FES.
- `--fes-level=2` is a more aggressive version, which can in some cases favor cache usage.

--array-static-indexing, -asi

It may not be always possible for EVARIX™ to identify that the size of an array does not vary along the execution, in particular when individual array elements are referenced through array indexing. Such arrays are classified by EVARIX™ as *dynamic*, which precludes some optimizations and the parallelization of `for` loops. This option allows to specify that array sizes are never modified through array indexing. This must be used carefully, because if it happens to be false, the generated code may be erroneous.

2.5.4 Parallelization options

--enable-parallel-pragmas, -epp

This options turns on manual parallelization features, relying on parallelization annotations. This option cannot be used in conjunction with `--auto-parallelization-level`.

--auto-parallelization-level=<level>

EVARIX™ features automatic parallelization capabilities, which can be turned off/on by adjusting the automatic parallelization level:

- `--auto-parallelization-level=0` (default) turns off automatic parallelization.
- `--auto-parallelization-level=1` turns on the automatic parallelization of vector/array operations when possible, depending on a cost function which must be below the threshold specified by -–parallel-for-threshold.
- `--auto-parallelization-level=2` (linux only) turns on the automatic parallelization of well-formed `for` loops.
  See also `--parallelization-report-level` and `--array-static-indexing`.
- `--auto-parallelization-level=3` (linux only) turns on the automatic parallelization of vector/array operations and well-formed `for` loops.
  See also `--parallelization-report-level` and `--array-static-indexing`.

In gpu sections (see 4.8.3), identified parallel element wise operations or parallel loop nests are turned into gpu kernels. Otherwise, the generated loop nests are tagged with OpenMP annotations.

In all cases, remind that automatic parallelization is no magic, and relies on advanced internal analyses and subtle trade-offs between compilation speed and precision, and may not be able to uncover existing parallelism or decide whether parallelization will bring more performances in all cases. In these cases, you may consider using the `--enable-parallel-pragmas` options for manual parallelization using annotations.

--parallelization-report-level=<level>

When automatic parallelization is turned on (see --auto-parallelization-level), EVARIX™ default behavior is to silently parallelize array operations and loops. The `--parallelization-report-level` can be set to get some information about the parallelization process:

- `--parallelization-report-level=0` (default) silent parallelization.

- `--parallelization-report-level=1` For each loop which is a valid candidate for automatic parallelization[1], displays whether it is parallelizable and under which conditions, or, if not, the reasons precluding its parallelization. These reasons may be scalar variable dependences, or array dependences. In both cases the names of the variables responsible for the inter-iterations dependences are given. If a loop is deemed parallelizable, it also reports if it is finally generated as parallel depending on eVariX™ parallelization policy.
- `--parallelization-report-level=2` (advanced level) In addition to the features of level 1, level 2 displays more information about array dependences in the form of unions of convex polyhedra corresponding to the sets of array elements responsible for the inter-iterations dependences.

--parallel-for-threshold=<threshold>

Sets the maximum threshold for the cost functions limiting the automatic parallelization of array operations (see also `--auto-parallelization-level=1`). Default value is 1000.

--gpu-codegen-policy=<level>

When manual parallelization is set with `--enable-parallel-pragma`, and gpu annotations are present in the source code, this option sets the GPU communication generation optimization policy used by eVariX™. The integer parameter can be 0, 1 or 2, default value is 1. The greater the `<level>`, the more optimizations are applied.

--gpu-double-precision

By default, in order to favor speed over precision, the code generated by eVariX™ uses single precision floating point data and arithmetic on GPU, even if the default is double precision in MATLAB®. However, this may lead to results which are different from the original code. Use this option to enforce double precision floating point data and computations on gpu.

--gpu-log

When the `--gpu-log` option is used for the generation of an executable, eVariX™ adds conditional log statements to the generated code. Logs can then be turned on by adding an optionnal argument to the generated executable:

```
my_exe --gpu-log-level=<ARG>
```

where `<ARG>` can be either `WRN` or `NFO`. `WRN` gives higher level information about the gpu execution; `NFO` adds some logs about communications and kernel compilation.

When the `--gpu-log` option is used for the generation of a mex library, the logs are systematically turned on at the `NFO` level, and this cannot be changed. The only way to turn logs off in the case of mex generation is to invoke eVariX™ again without the `--gpu-log` option.

Beware that the kernel compilation messages only mean that these operations have been requested, not that any job has actually been performed as compiled kernels are cached for later reuse.

2.5.5 Advanced options

--input-information

Display detailed information about the input source code (number of functions, of lines of code, etc).

--inline-max-stmt-nb=<max>

Set the maximum number of statements in caller after inlining a function call. `max` must be an integer, default is 199.

[1] A valid candidate is a `for` loop with a scalar integer index such as `for i = 1:n`, and not inner `break` statement.

--inline-max-weight=<max>

Set the maximum weight of the caller after inlining a function call. The weight is a function of the number of statements and of internal parameters. `max` must be an integer, default is 199.

--log-file=<file>

Log all streams of COLD© into `file`.

--math-optims, -mo

This experimental option turns on some mathematical optimizations. Depending on your code, it may give some additional speed-up.

--pragma-entry=<entry name and types>

Specify entry point and its argument types. This is an alternative to the `entry` code annotation.

--print-crt-default

Display the default version of the CRT used by COLD©.

2.6 Configuring eVariX™

The behavior of `evarix` is partly driven by a configuration file, by default `EVARIX_INSTALL_DIR/etc/evarix.cfg` on Linux, and `EVARIX_INSTALL_DIR\etc\evarix.cfg` on Windows. It specifies the compilers used by the different steps, their options, and the libraries used for the link step. The behavior of `evarix` can be tailored by editing the configuration file or by specifying a different configuration file using the `--configuration-file` option. A specific configuration can also be chosen using the `--select-configuration` option (see Section 2.4).

The remainder of this section describes the content of the configuration file, which is made of a series of *blocks* specifying part of the configuration information. Each block is in turn a list of key/value pairs separated by an '=' sign.

The file is organized in several sections:

- The preamble specifies the default compilers, and defines several configurations (see Section 2.6.1).
- The compiler section specifies the different possible compilers configurations (Section 2.6.2).
- The libraries section finally describes the libraries configurations (Section 2.6.3).

2.6.1 Configuration blocks

Defaults

The default compilers are identified by

```
Default<TYPE>Compiler = NAME
```

where <TYPE> is `CPP`, `Cold` or `MEX`; and `NAME` the name of the compiler in the configuration file.

Configurations Groups libraries link order

A *configuration* specifies some of the libraries required to compile the C++ code generated by Cold©. Each *configuration* belongs to a unique *configuration group*. For instance, the `language` configuration group contains the configurations specific to the languages supported by eVariX™.

The `ConfigurationsLibsOrder` field defines the dependency order between the libraries specified by the different configuration groups, that is the order[2] in which the libraries will be passed onto the linker. In case of cyclic dependencies between two library groups, they must be surrounded by curly braces. For instance, the following definition:

```
ConfigurationsLibsOrder = lowlevel {language env} graphic
```

specifies that the `language` and `env` configuration groups libraries may depend on each other.

Configuration

Each configuration block is composed of several fields:

- **Configuration**: defines the name of the configuration, prefixed by it configuration group name (such as `language:matlab`.
- **Required**: the libraries (see 2.6.3) required to link the C++ code generated by Cold©.
- **Compulsory**: This optional field can take only one value (`true`), and is used to specify that the configuration is always required to link the code generated by Cold©.

2.6.2 Compilers blocks

A compiler block is used to define a compiler default options. There are three kinds of compiler blocks: Cold© and CPP and MEX.

[2] In fact the reverse order, as highest level libraries must named first.

Cold© Compiler: The Cold© compiler block defines the Cold© compiler and its options, and has three fields:

- **ColdCompiler**: the name of the Cold© compiler.
- **Path**: the path where the compiler can be found.
- **Flags**: the default flags for the Cold© compiler.

Note that the **Path** field is not required if the Cold© compiler can be found within the `PATH` environment variable.

CPP Compiler: A CPP compiler block defines a C++ compiler and its options, and may have several fields:

- **CPP**: the name of the compiler.
- **Flags**: the flags to pass on to the compiler. On Linux, this field must contain at least the `-std=c++11` flag.
- **ReleaseFlags**: the flags to pass on to the C++ compiler in release mode. Note that *release* is the default compilation mode.
- **ParallelFlags**: the flags to pass on to the C++ compiler when using automatic parallelization or parallelization pragmas.
- **MexFlags** (linux only): the flags passed on to the C++ compiler by the MathWorks MEX compiler. On Linux, this field must contain at least the `-std=c++11` flag.
- **MexParallelLibs** (linux only): the flags passed on to the C++ compiler by the MathWorks MEX compiler when the C++ code is parallel.
- **Libs**: the common libraries used by the C++ compiler. By default, the C++ code must be linked with the C++ math library (-lm).

By default, `evarix` uses the g++ compiler on Linux systems and MinGw on Windows systems. The default compiler can be specified using the key **DefaultCPPCompiler**. The value must be the name of a compiler block.

MEX Compiler: The MEX compiler block defines a MEX compiler and its options (linux only).

- **MexCompiler**: the name of the MEX compiler.
- **CPP**: the default C++ compiler called by the MEX compiler.
- **Flags**: the flags for the MEX compiler.

2.6.3 Libraries blocks

Each library block defines a library which has to be linked to the C++ code generated by Cold©. Such a block can have the following fields:

- **Library**: the name of the library.
- **Include**: the directory containing the header files of the library.
- **Libs**: the library(ies) to link with the C++ code in release mode.
- **Path**: the path to the library(ies).

CHAPTER 3

IMPORTING EXTERNAL LIBRARIES (LINUX ONLY)

eVariX™ offers the possibility to generate code that calls functions from external C/C++ libraries, much as MATLAB® allows to execute mex libraries. The process requires that you provide:

- a `library file`[1] defining one or several fonctions (or *interfaces*) taking as formal parameters CRT© data types; these functions must convert the CRT© data to your library types, call your library functions, and convert back your output data to the CRT© types. The CRT© provide functions to ease the conversion from CRT© arrays to regular C arrays.
- a `library description file`, which contains one or several `extern` annotations describing the *signatures* of the interfaces with a syntax very close to the MATLAB® syntax.

Both files must be in the same directory and bear the same name, except for the file extension. For instance, if your library description file is named `libmyfunc.m`, your library file must be named `libmyfunc.a` or `libmyfunc.so`.

The next sections define how to write a library description file, how to write and generate the library containing the interfaces, and finally how to generate an executable calling the library interfaces from a MATLAB® script.

Throughout this chapter, we will use two trivial examples:

1. the first library, `libintsum.m`, provides a function that takes as inputs an integer `n` and matrix of integers, and returns an integer;
2. the second library, `libsumprod.m`, defines a function that takes two real matrices as inputs, and returns two other matrices.

3.1 The library description file

The library description file must be named after the library you wish to import, with the `.m` extension. It defines the *signatures* of the functions defined by the library, using one or several `extern` annotations. A signature defines the relation between the input types accepted by the function, and its output types.

For instance, to describe the function provided by `libintsum.m`, the syntax is:

```
%pragma evarix extern int = intsum(matrix{int}, int)
```

It means that when a matrix of integers and a scalar integer are provided to `intsum`, then the type of the result is a scalar integer.

A call to this function in your main script can be:

[1] Either a `.a` or `.so` file on linux.

```
A = [1 2 3; 4 5 6];
s = intsum(A,4);
```

Beware that the default type in MATLAB® is double precision reals. However, EVARIX™ tries to more precisely infer scalar types, so you may need to provide interfaces for other precisions. In the previous code, EVARIX™ is able to find out that `A` is always a matrix of integers. So you must provide a description (and a function!) matching this type.

To describe the function provided by `libsumprod.m`, the syntax is:

```
%pragma evarix extern [matrix{real},matrix{real} = sumprod(matrix{real}, matrix{real})
```

A call to this function in your main script can be:

```
A = [1.1 2.1 3.1; 4.1 5.1 6.1];
B = [10.2 20.2 30.2; 40.2 50.2 60.2];
[C,D] = sumprod(A,B);
```

In MATLAB® as well as in EVARIX™ runtime library, vectors are represented as matrices where one dimension is equal to one. As such, the previous signature is perfectly valid to handle vectors:

```
U = [1.1 2.1 3.1];
V = [10.2 20.2 30.2];
[W1,W2] = sumprod(U,V);
```

However, during its type and shape propagation, EVARIX™ will consider `W1` and `W2` as matrices, and not as vectors. This may lead EVARIX™ to lack some information to infer precise enough shapes for subsequent operations. In that case, you can provide additional signatures for a single function prototype. For instance, you can tell EVARIX™ that when `sumprod` is given two row vectors as inputs, its outputs are also row vectors. And similarly for column vectors:

```
%pragma evarix extern [row{real},row{real} = sumprod(row{real}, row{real})
%pragma evarix extern [col{real},col{real} = sumprod(col{real}, col{real})
```

EVARIX™ can import several functions from a single library, if the library description file provides the corresponding signatures. The library name does not need to match the names of the functions. For instance, the following could be the content of a library description file named `libmyfunc.m`:

```
%pragma evarix extern [row{real},row{real} = foo(row{real}, row{real})
%pragma evarix extern [col{real},col{real} = foo(col{real}, col{real})
%pragma evarix extern [matrix{real},matrix{real} = foo(matrix{real}, matrix{real})

%pragma evarix extern hello()
%pragma evarix extern hello(string)
```

Notice that the two last signatures correspond to two different functions returning void.

⚠ Warning

The library can define functions bearing the name of MATLAB® primitives, such as `sum` for instance. In this case, the library definition overrides the MATLAB® definition, and all references to the function `sum` in your MATLAB® model must correspond to a signature provided in your library. eVarix™ systematically issues a warning.

3.2 Writing and compiling the library

The library source file containing the interfaces must be written in C++. This section describes first how to write the function prototypes, and then how to convert their input parameters, which have CRT© data types, to and from your own types.

3.2.1 Writing interface function prototypes

For the sake of simplicity, each interface function must follow the following rules:

1. an interface function always returns `void`;
2. the output parameters declared in the `extern` annotation are the first input parameters of the interface, in the order in which they appear in the annotation (and in the function calls in your matlab model); they are always C++ non-const references; character strings are declared as `char*&`;
3. the input parameters declared in the `extern` annotation come after the output parameters, also in the order in which they appear in the annotation (and in the function calls in your matlab model); they are always C++ const references, even for numerical scalar inputs, but except for character strings, which are declared as `const char*`.

For the `intsum` function declared above, the C++ function prototype is:

```
void intsum(int& out
            , const crt::Array<int,2,crt::heap_allocator>& in1
            , const int& in2)
```

If your script has parts running on a gpu, then you must provide an interface using the smart data, even if the function is not called from the gpu part:

```
void intsum(int& out
            , const crt::ocl::vector<int,int,2,crt::heap_allocator>& in1
            , const int& in2)
```

For the `sumprod` function, the function prototype corresponding to the annotation is:

```
void sumprod(crt::Array<double,2,crt::heap_allocator>& out1
             , crt::Array<double,2,crt::heap_allocator>& out2
             , const crt::Array<double,2,crt::heap_allocator>& in1
             , const crt::Array<double,2,crt::heap_allocator>& in2)
```

And for the `hello` functions, the prototypes are:

```
void hello();
void hello(const char*);
```

f you are unsure about how to write your function prototypes, run EVARIX™ on your MATLAB® code with the `--keep` option, and either with the `importlib` annotation or using the `--import-lib` option (see Section **??**). The necessary interfaces are declared with the keyword `extern` at the beginning of the generated C++ file.

3.2.2 Converting CRT© arrays to and from C arrays

The CRT© arrays which are the input parameters of your interfaces should not be modified directly, as their internal state is maintained by the CRT©. You should always manipulate them through the CRT© functions.

This section demonstrates how to get the dimensions of the arrays and a copy of their buffer in C-style arrays, and conversely, how to set the state of the output arrays from C-style arrays.

First, as the code uses the CRT© data types, it is necessary to include some CRT© header file:

```
#include "crt_matlab.h"
```

If your script has parts running on a gpu, then you must also include a smart data specific header file:

```
#include "ocl/matlab/_crt_ocl_matlab.h"
```

Let us then consider the input array `in1` of the function prototype `sumprod`. To get your own safe C-style copy of `in1` and its dimensions, you must use the function `toCArray`:

```
double *c_in1 = nullptr;
size_t *dims1 = nullptr;
toCArray(c_in1, dims1, in1);
```

It takes as input a reference to a pointer to the type of the input array (here `double`), a reference to a pointer to `size_t`, and the input array itself. After the call to `toCArray`, `c_in1` contains the same data as `in1`, and, as the latter is a two dimensional array, `dims1` is an array of 2 elements representing the sizes of its two dimensions.

Notice that the CRT© arrays, as MATLAB® arrays, are column-major arrays. If your library operates on row-major arrays, you must convert them. EVARIX™ provides an interface to directly get a row-major version, by adding a parameter to the call to `toCArray`:

```
double *c_in1 = nullptr;
size_t *dims1 = nullptr;
toCArray(c_in1, dims1, in1, crt::ROW);
```

There is no difference if your script has parts running on a gpu.

For output arrays, the mechanism is just the reverse, except that you are responsible for the allocation and initialization of the output C-style buffers, and the arrays of `size_t` holding the dimensions. A constructor of the `crt::Array` class lets you perform the conversion.

```
size_t dims_out1[2] = ... ;
size_t nout1 = dims_out1[0]*dims_out1[1];
double* c_out1 = (double*) malloc(sizeof(double)*nout1);

// ... your code here which initializes c_out1

out1 = crt::Array<double,2,crt::heap_allocator>(c_out1
                                                , dims_out1
                                                , (size_t)2);
```

If `c_out1` is a row majour array, then you can convert it to a column major array:

```
out1 = crt::Array<double,2,crt::heap_allocator>(c_out1
                                                , dims_out1
                                                , (size_t)2, crt::ROW);
```

If your script has parts running on a gpu, then you must use the smart data constructor:

```
out1 = crt::ocl::vector<double,double,2,crt::heap_allocator>(c_out1
                                                             , dims_out1
                                                             , (size_t)2);
```

If you want to update only some values of `out1`, you must first initialize `c_out1` as for an input array, then work on your copy, and finally assign your data back to the output array:

```
size_t* dims_out1 = nullptr;
double* c_out1 = nullptr;
toCArray(c_out1, dims_out1, out1);

// ... your code here which modifies c_out1

out1 = crt::Array<double,2,crt::heap_allocator>(c_out1
                                                , dims_out1
                                                , (size_t)2);
```

When you use the `toCArray` function and the `crt::Array` class constructor presented above, there is no sharing between the allocated C buffers and the CRT© data. So you must free your data at the end of the interface function. Don't forget the arrays holding the dimensions:

```
free(c_in1);
free(dims1);
...
free(c_out1);
free(dims_out1);
...
```

3.2.3 Single precision

The default type for MATLAB® array elements is double. If your library works on single precision values, you can provide an interface using the `float` data type, and explicitely convert the data to single precision in your MATLAB® script before invoking your library function. For instance, an interface declaration could be:

```
%pragma evarix extern matrix{single} = foo(matrix{single})
```

and the corresponding function prototype:

```
void foo(crt::Array<float,2,crt::heap_allocator>&
         , const crt::Array<float,2,crt::heap_allocator>&)
```

To use it in your MATLAB® code:

```
A = single([1.1 2.2 3.3; 4.4 5.5 6.6]);
B = foo(A);
```

3.2.4 Compiling the library

To generate the library archive you must compile its source files and link them against the CRT© libraries. For that purpose, you need to specify the path to the CRT© header and library files. We provide below a sample cmake file to achieve this purpose. It has two options, to specify the CRT© installation directory (usually your eVariX™ installation directory) and the version of the CRT© you use (for eVariX™ 2.6.0, it is 2.6.3). The library is installed in the same directory as the source file.

```
cmake_minimum_required (VERSION 2.8)
project (IMPORTLIB)

set (CMAKE_CXX_COMPILER "g++")
set (CMAKE_CXX_FLAGS "-std=c++11 -Wall -Werror")

option (CRT_DEFAULT_DIR "CRT Default Directory"   OFF)
option (CRT_DEFAULT_VERSION "CRT Default Version"   OFF)

if (NOT CRT_DEFAULT_DIR OR NOT CRT_DEFAULT_VERSION)
  message (FATAL_ERROR
     "this project requires a CRT Default Directory and a CRT Default version")
endif ()

include_directories("${CRT_DEFAULT_DIR}/include/${CRT_DEFAULT_VERSION}/")
link_directories("${CRT_DEFAULT_DIR}/lib/${CRT_DEFAULT_VERSION}/")

add_library (mylib mylib.cpp)
target_link_libraries(mylib crt_m crt crtdefault)

install(TARGETS mylib DESTINATION ${CMAKE_CURRENT_SOURCE_DIR})
```

If your library is parallelized using OpenMP, then it must be linked against the same OpenMP library as the code generated by EVARIX™, `libiomp5.so`, located in

`${CRT_DEFAULT_DIR}/lib/${CRT_DEFAULT_VERSION}/mkl/`.

⚠ Limitation

If your script has parts running on a gpu, then your library functions must not run on the gpu.

3.3 Using the library

There are two possibilities to tell EVARIX™ to use an external library:

1. the `importlib` annotation
2. the `--import-lib` option

The `importlib` annotation must be placed in the main script file before any use of the functions it declares. It is thus a good pratice to place it at the head of the file. For instance, to use an external library described in `mylib.m`, and in the directory `/path/to/mylib`, the annotation is:

```
%pragma evarix importlib (/path/to/mylib/mylib.m)
```

You can use several `importlib` annotations, provided that the libraries define functions bearing different names.

The `--import-lib` option can alternatively be used on the EVARIX™ command line:

```
evarix myscript.m -o myscript --import-lib=/path/to/mylib/mylib.m
sample
```

CHAPTER 4

WRITING EVARIX™ COMPLIANT CODES

This chapter describes the basic MATLAB® features supported by EVARIX™, as well as language extensions specific to EVARIX™ and provided as annotations (see section 4.8). The list of the supported MATLAB® operators and functions is given in Part II.

4.1 Introduction to compiling with eVariX™

EVARIX™ core component is a compiler, COLD©, producing C++ code from *correct* MATLAB® code[1], which chains six main phases:

- the *parser* reads the MATLAB® code and transforms it into a convenient internal format. This phase reports syntactic errors.
- the *compliance analyzer* determines whether the syntactic features used in the input code are supported by EVARIX™ or not. This phase reports as errors the features which are wholly unsupported, and as warnings those that prevent EVARIX™ from generating efficient code, or that disable some advanced optimization phases without stopping the code generation process.

  For example, the following code uses several unsupported features to dynamically [2] generate a new variable called `foo`. The code is split into two files:

  – `uselessFunction.m`

```
function varName = uselessFunction()
    varValue = zeros(10);
    varName  = 'foo';
    eval(sprintf('global %s;\n %s=varValue;\n', varName, varName));
end
```

  – `code.m`

```
varName = uselessFunction();
eval(sprintf('global %s',varName));
foo % display an array of 10x10 of 0
```

  The compliance analysis produces the following error report:

[1] "*correct*" means here that the code executes without any error in the MATLAB® environment, and that it follows the *compliance rules* described in this chapter. Otherwise, the behavior of EVARIX™ and of the generated code may be undefined.
[2] meaning during the execution.

```
------ COLD ERROR - FRONT END -------------------------------------------------

  Failed in pass: ComplianceAnalysis (Compliance analysis: check that all features
                  used in the code are supported by the compiler.)

Script part
at code.m:1
  Undefined Symbols:
    foo  (at line:3)
  Unsupported functions:
    eval  (at line:2)

Function uselessFunction
at ./uselessFunction.m:1
  Unsupported functions:
    eval  (at line:4)


------ COLD COMPILATION PROCESS SUMMARY ----------------------------------------
  STEP 1  FRONT END:                              FAILED
  STEP 2  PRE-INFERENCE OPTIMIZATIONS:            NOT REACHED
  STEP 3  INFERENCE:                              NOT REACHED
  STEP 4  POST-INFERENCE OPTIMIZATIONS:           NOT REACHED
  STEP 5  SEMANTICAL TRANSFORMATIONS:             NOT REACHED
  STEP 6  LOW-LEVEL OPTIMIZATIONS:                NOT REACHED
  STEP 7  CODE GENERATION:                        NOT REACHED

-------------------------------------------------------------------------------
```

This report identifies the erroneous phase and pass (in this example, the compliance analysis pass in the front-end phase). The errors are then listed by functions and by groups of errors:

- in the *script* part (file `code.m`) there are two kinds of errors :
  - a call to the unsupported function `eval`;
  - the use of the symbol `foo`, which is known by MATLAB® during the execution of the code because it is defined by the call to the `eval` function, but which is unknown to EVARIX™.
- in the function `uselessFunction` (file `uselessFunction.sce`) there is a call to the `eval` function.

The end of the report is a summary of the different compilation phases and their status.

- the *type inference* mechanism, which may report additional errors, and which is more extensively presented in the next Section (4.2).
- a series of *semantical* transformations.
- a series of *optimization* phases.
- and, finally, the *code generator* in charge of writing the C++ code.

Notice however, that some errors and un-compliances are not detectable at compile time.

Note

The input code is assumed to be correct and eVariX™ compliant.
Otherwise, the behavior of EVARIX™ and of the generated code may be undefined.

4.2 Types

In MATLAB®, the *types* of the program variables are dynamically determined during the execution. In particular, this implies that the type of a variable can change during the execution, as illustrated by the following example:

```
A='foo';
A=[1 2 3 4];
A=[1i 2i ; 3 2+4i]
```

After the execution of the first line, the variable `A` refers to a string; after the second, to an array of integers of size 1×4; and after the third to an array of complex integers of size 2×2.

On the contrary, the C++ language provides a strong typing scheme (the type of a variable is static and known at compile time) which is key to performance. To bring this performance to your application, it is therefore necessary that EVARIX™ be able to determine a type for each of the program variable, without executing the program: this is called *type inference*. Therefore, EVARIX™ enforces a few programming rules to ensure that this process can be achieved. Some dynamicity is allowed, but remember that the more accurately the types can be determined, the more performance can be brought to the generated code. These rules are presented in Section 4.2.2. But before that, let us define some vocabulary to avoid ambiguities.

4.2.1 What's in a type?

We call a **basic type** either a **numeric basic type** (boolean, integer, real, complex), or a character **string**.

Then, the **type** of a variable is considered by EVARIX™ as the combination of two characteristics:

- the **element type**; for a scalar variable, it is its basic type; for an array, it is the basic type of its elements;
- the **shape**; for a scalar variable, it is a **scalar shape**[3]; for an array, the shape is composed of
  - the number of dimensions;
  - the number of elements on each dimension;
  - optional additional properties such as the sparsity.

The **size** of an array collectively designates its number of dimensions and the number of elements on each dimension.

4.2.2 Type inference rules

EVARIX™ supports to some degrees the dynamicity of variable values. This section presents the rules which must however be followed by the input source code to allow the type inference step to determine a type for each variable. In addition, being aware of the rules which drive the typing mechanism will allow you to write programs which will lead to more efficient EVARIX™ generated executables (see also the next section, about typing and storage).

The following rules restrict the *element type* as well as the *shape* of a variable along its *life duration* or *scope*. This concept is defined in Section 4.2.4.

- **The element type of a numeric variable is the *maximum* of the element types of the expressions which are assigned to it**. If a variable is assigned *integers* and *reals*, then EVARIX™ infers its element type as being *real*. Similarly, if a variable is assigned *reals* and *complex reals*, then its inferred element type is *complex real*. For example, the following code snippet is EVARIX™ compliant:

[3] EVARIX™ differs here from MATLAB®, as the latter considers a scalar variable as a 1-by-1 array.

```
A = 1.1
A = 1
A = 1.1+2.2i
B = [1 2 3]
B = [i 2i 3i]
```

The type inferred for `A` is a scalar of *complex reals*, and the type for `B` is a vector of 3 *complex reals*.

- **String and numeric element types cannot be mixed for the same variable**:

```
A = 'foo'
A = 5
```

is not EVARIX™ compliant[4].

- **The number of dimensions of a variable cannot change**[5]: if a single value is assigned to a variable, for example `A=5`, the variable is considered as having a scalar shape for all its lifetime; if an array is assigned to a variable, its number of dimensions is constant over the whole variable life duration. For example, the following code is not EVARIX™-compliant:

```
A = ones(2,2)
A = rand(2,3,4)
```

because the number of dimensions of `A` is 2 after the first statement, and 3 after the second statement.

However, the promotion of a scalar shape variable to an array is supported. In this case, the variable has the number of dimensions of the array, but the number of elements in each dimension becomes equal to 1 when the variable holds a single value during the execution of the program.

Similarly, an array dimension size may dynamically become equal to 1 during the execution of the program. However, in this case, EVARIX™ will not be able to generate calls to the functions which would be the most adapted to the dynamic type.

4.2.3 Types and storage

The precision of the complete type attributed to an array-shaped variable also determines how it will be stored in memory by the generated code. EVARIX™ has two schemes for storing arrays:

- **Static arrays** are allocated in the internal stack of the CRT©, if EVARIX™ can prove that the variable size will not change during the execution of the program. Using this kind of array allows to get the best performances: the allocation is costless, there is no need to test for array bounds violations, and there is no need for reshaping. However, determining at compile time that the size of the array is constant is difficult, or even impossible. For now, EVARIX™ uses static arrays for variables fulfilling the following conditions:
  - the array is allocated by calling an initialization function from which the size can be deduced (`zeros`, `ones`, ...);
  - the write accesses to the array elements are indexed by constant integers and are inferior to the sizes of the corresponding array dimensions; or the `--array-static-indexing` option is used to guarantee that accesses through indexing never exceed the current array size;

[4] This is consistent with the previous rule as there is no *maximum* for a string basic type and an integer basic type.
[5] In the maximum scope in which it is defined and used, see Section 4.2.4

  - if the variable is overwritten by another expression, the sizes of all the dimensions of this expression are constant integers equal to the previous sizes of the array.

- In all other cases, EVARIX™ safely uses **dynamic arrays** whose size can change during the execution. The allocation of such arrays is always done on the *heap* and write accesses to the array are preceded by validity checks. If the index range of the access is greater than the current size of array, the array is reshaped [6]. This storage is thus less efficient than the static storage.

For example, in the following function where `n` is a scalar integer whose value is not numerically known at compile time, the array `A` will be allocated statically, whereas the arrays `B` and `C` will be dynamic arrays.

```
function foo(n)
    A = zeros(2);
    A(1,1) = 1;
    A
    B = zeros(10);
    B(1,n) = 1;
    B
    C = zeros(n);
    C(1,1) = 1;
    C
end
```

The array `B` will nevertheless be allocated statically if the `--array-static-indexing` option is used. But the user is then responsible for ensuring that the value of `n` is always less than or equal to 10, otherwise the execution will fail or give erroneous results.

4.2.4 Types and scopes

All the typing rules are relative to the **maximum scope** in which a variable is defined and used. In the following code snippet:

```
if (10 < 100)
  a = 10;
  disp(a);
else
  a = 'hello';
  disp(a);
end
disp('the end');
```

the maximum scope in which the first value assigned to `a` is used is the first branch of the `if` statement, which is also the scope of the assignment; this value is neither used in the second branch, nor in the statements after the termination of the `if`. Similarly, the maximum scope in which the second value assigned to `a` is used is the second branch of the `if` statement, which is also the scope of this second assignment. In each of these scopes, the type of `a` is constant, and EVARIX™ can determine that two variables are actually necessary, one for each of the `if` branch.

On the contrary, in the next example, only slightly different from the preceding one, the value of `a` is used in the statements after the termination of the `if`:

[6] If the access is valid, following the MATLAB® rules

```
if (10 < 100) then
  a = 10;
  disp(a);
else
  a = "hello";
  disp(a);
end
disp(a);
```

So, the maximum scope in which `a` is defined and used is the whole code extract. But, as the types of the expressions assigned to `a` are not compatible, this code is not EVARIX™ compliant, and this is reported as an error.

The support of global variables is still experimental. It is recommended to pass them as explicit parameters to all the functions in which they are defined or used. It must be noticed that using global variables may also prevent some optimizations, and always prevents the automatic parallelization of the `for` loops which have written effects on them, either directly or indirectly through function calls.

⚠ Limitation

- Global variables should be avoided because they can introduce ambiguity during symbol resolution (during the parsing, this step determines the *kind* of a symbol: function, variable, keyword, ...) : since the symbol resolution of EVARIX™ is static (whereas the one of MATLAB® is dynamic), EVARIX™ is not always able to distinguish between a global variable and a local variable sharing the same name.
- Persistent variables are not supported.

4.2.5 Specifying types and shapes

Several annotations are provided to specify either the element type of a variable, or the complete types of the input parameters of a function. This section defines the syntax of the type-specifying part of these annotations.

Element types EVARIX™ support the following basic types:

- boolean types: `bool`;
- integer types: `int`, `int8`, `uint8`;
- real type: `double`;
- complex type: `complex`;
- character string type: `string`.

Shapes Specifying shapes is done with the following syntax (where `T` is an element type):

- scalar shape: `<T>`;
- row vector: `row{<T>}`; `row{<T>, n}` if the vector has an integer constant size `n`.
- column vector: `col{<T>}`; `col{<T>, n}` if the vector has an integer constant size `n`.
- matrices (2D-array): `matrix{<T>}`. Use `matrix{<T>, m, n}`, where `m` and `n` are integer constants, if the matrix has fixed-sized dimensions;
- ND-arrays: `array{<T>,N}` where `N` is the number of dimensions of the array.

⚠ Limitation

Arrays of strings are not supported in this version.

4.3 Expressions

4.3.1 Constants

The following constants of MATLAB® are supported by EVARIX™:

Symbol	Meaning
`eps`	floating-point relative accuracy
`i`	imaginary part
`Inf`	infinity
`j`	imaginary part
`NaN`	not a number
`pi`	π

4.3.2 Colon notation

Colon notation is used to define vectors, array subscripts or `for` clauses. The syntax of colon notation is:

- $a : b = [a,\ a+1,\ a+2,\ \ldots,\ b]$ for $a < b$, a, b scalars;
- $a : b = []$ for $a >= b$, a, b scalars;
- $a : b : c = [a,\ a+b,\ a+2b,\ \ldots,\ a+b*d]$, for a, b, c scalars and $d = \lfloor \frac{c-a}{b} \rfloor$;
- $a : b : c = []$, for a, b, c scalars if $b == 0$ or $b < 0$ and $a < b$ or $b > 0$ and $a > b$.

4.3.3 Array Definition

Matrices (2D arrays) and ND-arrays containing boolean, integer, real or complex values can be defined using brackets. Columns are separated by a comma (,) or a space (' '). Rows are separated by a semicolon (;) or by starting a new line.

```
l = [1 2 3];                    % vector of integers

m = [1.1 2.2 3.3; 4.4 5.5 6.6]; % 2*3 array of reals

n = zeros(1,2,2);               % 1*2*2 array of reals
n(1,1,1) = 1;n(1,2,1) = 3;
n(1,1,2) = 2;n(1,2,2) = 1;
```

⚠ Limitation

Array definitions mixing line continuations and comments are not supported and may lead to weird EVARIX™ behavior.

4.3.4 Array Indexing

Arrays can be indexed by:

- scalar integers;
- matrices of integers;
- colon notation;
- logical expressions.

```
A=[1.1 2.2 3.3 4.4 5.5];
B=[2 3 4];
A(3)          % Display 3.3
A(:)          % Display 1.1 2.2 3.3 4.4 5.5
A(2:4)        % Display 2.2 3.3 4.4
A(3:end)      % Display 3.3 4.4 5.5
A(3:2)        % Empty matrix
A(3:-1:2)     % Display 3.3 2.2
A(end-1:-2:1) % Display 4.4 2.2
A(B >= 3)     % Display 2.2 3.3
```

4.3.5 Array concatenation

⚠ Limitation

The only supported concatenation is the concatenation of matrices. The concatenation of multidimensional arrays is not yet supported.

Thus the following code is not EVARIX™-compliant:

```
a = zeros(2,3,4);
b = zeros(2,3,4);
[a b]
[a;b]
```

4.3.6 Structures

Structures are variables than can contain data of varying types and sizes. EVARIX™ supports MATLAB® structures containing the following types of data:

- constants:

```
A.B = 5;
A.C = 'bar';
A.D = complex(4.6,8);
% The variable A is a structure containing 3 fields: 'B' 'C' 'D'
```

- matrices (2D arrays):

```
A.E = [1 2 3 4; 5 6 7 8];
A.F = zeros(10, 10);
```

- other structures:

```
A.G.A = 5;
A.G.B = zeros(5,5);
```

The `struct` function can also be used to initialize a structure.

Each field of a structure is considered as a normal variable, and therefore all rules described in this document apply.

⚠ Limitation

- Arrays of structures `A(1,1).B` are not yet supported.
- Using structures is more costly than using scalars or arrays. If you experience performance issues, please consider using the latter.
- **This feature is still in beta version. Please report any issue to** evarix.support@silkan.net**.**

4.4 Control Flow

EVARIX™ supports the following control flow constructions.

4.4.1 Conditional expression

The syntax of the conditional expression is:

```
if condExpr
...
elseif condExpr2
...
else
...
end
```

where the $\texttt{condExpr}_i$ are boolean (true or false) expressions.

4.4.2 for loop

The syntax of the `for` loop is:

```
for index=expression
   % statements
end
```

where `expression` can be a vector (in particular, a vector resulting from colon notation) or a 2D-array. In the latter case, the index of the loop is a column vector from subsequent columns of the array on each iteration. The keywords `break` and `continue` are supported within `for` loops, but `break` prevents the automatic parallelization of the loop.

⚠ Limitation

- 3D-arrays are not supported in for clauses.
- The syntax `for (index=expression) .. end` (parentheses around the clause) is not supported.
- The value of the index of the loop can be used after the loop. However, if the loop is not executed, the value of the index is the first value of `expression`, and not the empty array `[]`.

4.4.3 switch / case block

The syntax of the `switch/case` block is:

```
switch expr
case val1
...
case valn
otherwise
...
end
```

where `expr` is a variable of scalar or string type and `vali` are integers or characters.

4.4.4 while loop

The syntax of the `while` loop is:

```
while condExpr
   % statements
end
```

where `condExpr` is a boolean (true or false) expression. The keywords `break` and `continue` are supported within `while` loops.

4.5 Functions

4.5.1 Syntax

A function begins with the keyword `function` and is ended by the keyword `end`. Any supported type can be passed or returned. A function can take or return any number of arguments.

```
function output = name(arguments)
  % statements
end
```

Note that in MATLAB® codes, scripts and functions must be defined in different codes files. Each function must be defined in its own file whose name must be the name of the function followed by the extension (.m).

⚠ Limitation

- Calling a script in another script is not yet supported.
- Variable number of arguments or returned data (`varargin` and `varargout`) are not supported.

4.5.2 Calling functions

The same function can be called with different types of arguments.

```
function [res]=foo(x)
    res = x + 1
end

y = foo(1)              % calling foo with an integer   => y = 2
Y = foo([1 2])          % calling foo with a 1x2 matrix => Y=[2 3]
Z = foo([1 2 3;4 5 6]) % calling foo with a 2x3 matrix => Z is a 2*3 matrix
```

⚠ Limitation

- **Calling functions in command mode** (e.g. `disp alpha`) is only supported when the command is followed by a name. It is therefore highly recommended not to use this feature, and to prefer its regular function call equivalent (e.g. `disp("alpha")`), because in most cases, EVARIX™ cannot distinguish between an expression and a command mode call and treats the latter as the former.

4.6 Classes

The definition of a MATLAB® class is made of a succession of blocks – `properties`, `methods`, `events` and `enumeration` blocks – embedded in a surrounding class definition block.

The class definition can be written in a single file which bears the name of the class with the `.m` extension. Class definition directories, where the whole class definition is spread over several files in a directory named after the class name, are not supported.

⚠ Limitation

- Class definition directories are not supported.
- `events` and `enumerations` are not supported

4.6.1 Class definition block

The syntax of a class definition block is the following:

```
classdef (ClassAttributes) AClass < SuperClass
 ...
end
```

`ClassAttribute` is a list of attributes with their values.

`SuperClass` specifies another class from which `AClass` inherits. A user class can inherit from another user class, or from a MATLAB® internal class. Currently, EVARIX™ supports only inheritance from the MATLAB® `handle` class.

⚠ Limitation

- Class attributes are currently ignored.
- Class inheritance is not supported, except from the MATLAB® `handle` class.

4.6.2 Properties

A class definition may contain one or several *properties* blocks, one for each unique set of `PropertyAttributes`. A *properties* block defines the *properties* held by each object, with their attributes and their initial default values. This set of properties is fixed: properties cannot be dynamically added to objects, except when the class derives from the MATLAB® `dynamicProps` class, which is not supported by EVARIX™.

The syntax of a properties block is the following:

```
classdef ClassName
 properties (PropertyAttributes)
    prop1
    prop2 = defaultValue2
 end
 ...
end
```

EVARIX™ does not support the definition of properties with no initial value. This is to ensure that type and shape inference are possible. When no initial value is specified, the default value is the empty array `[]`, which may not be compatible with further scalar values. Hence, only the form `prop2 = defaultValue2` is supported. If a property is meant to hold a structure or another class object, then its default value should be set to `[]`.

The initial values must be standard MATLAB® expressions, but they musn't reference any variable. MATLAB® specifies that these expression are evaluated only once, and that the resulting values are then copied into the newly created object properties. Currently, EVARIX™ does not store the properties initial values, which are computed anew at each class object creation. This forbids expressions with global effects, such as calls to random generators functions, or timing functions. It may also have a small impact on performances.

⚠ Limitation

- Properties initial values must always be specified.
- Properties initial values are evaluated at each object creation.
- Properties attributes are ignored by EVARIX™.

4.6.3 Methods blocks

A *method block* defines one or several methods of the class. There may be several methods blocks, one for each unique set of *MethodAttributes*. The syntax is the following :

```
methods (MethodAttributes)
    function obj = myMethod(obj)
    ...
    end
    ...
 end
```

There are several kinds of methods:

- *constructors*; they bear the same name as the class, and must return an object of the class:

```
function obj = ClassName(arg1,...)
     obj.PropertyName = arg1;
     ...
end
```

  A class can have at most one constructor.

- *ordinary methods*; they have the same syntax as an ordinary function, but their left-most input argument must be an object of the class on which the function is invoked:

```
function out = ordinaryMethod(obj,arg1,...)
     ...
end
```

- *static methods*; contrarily to ordinary methods, their left-most input argument is not an object of the class. They behave as ordinary functions, which are encapsulated in the class. Static methods are specified by giving the `Static` attribute to the method block.

⚠ Limitation

- The only supported *MethodAttribute* is the `Static` attribute. The other attributes are ignored.

4.6.4 Calling methods

MATLAB® uses priority rules to determine which method must actually be invoked. If it is invoked with the usual function calling convention (e.g. `foo(obj1, obj2)`), then MATLAB® calls the method of the left-most argument whose class is not *inferior* to other argument classes. However, this rule does not apply when methods are invoked with the *dot* notation:

```
Y = obj1.foo(obj2);
```

In this case, the method which is called is the function `foo` from the class of `obj1`. Only the *dot* notation is uspported by EVARIX™.

MATLAB® allows to define, for each property, special methods (*set* and *get* methods) which are called when object properties are accessed with the *dot* notation, in order to control the actions that may be performed. EVARIX™ does not support this behavior.

⚠ Limitation

- Methods can only be invoked through the *dot* notation.
- Accessing object properties through the *dot* notation does not trigger the invocation of user-defined properties *set* or *get* methods.

4.6.5 Class objects and type inference

The inference rules presented before still hold for objects and their inner properties. Hence, EVARIX™ must be able to statically determine a unique type for each variable that holds an object, and for each of its properties. In particular, the following codes cannot be handled by EVARIX™:

```
obj = MyClasse();
obj = MyOtherClass();
```

```
if (cond)
  obj = MyClass();
else
  obj = MyOtherClass();
end
obj
```

because no unique type can be inferred for `obj` in the minimal scope considered by EVARIX™.
Similarly, the following code is not supported:

```
% in file MyClass.m
classdef MyClass
  properties (SetStatus = private)
    n = 0;
  end
  methods
    function setValue(obj, m)
      obj.n = m;
    end
  end
end

% script part
t = MyClass();
t.setValue(10);
t
t.setValue('hello');
t
```

because the property `n` of the object `t` successively takes values of incompatible types: an integer as initial value and during the first call to `setValue`, and a character string in the second call to `setValue`.

4.6.6 Handle class objects

The `handle` class is an internal MATLAB® class, which modifies the behavior of objects so that no object copy is performed during object assignments, or when the object is passed as an input argument to a function (including an object class method)[7]. Beyond allowing performance gains, it also gives the possibility to accumulate values or states in a single object over several method calls.

A user class which inherits from the `handle` class thus inherits from this behavior. We call such a user class a *handle classe*, and its objects, *handle class objects*.

However, this behavior breaks some fundamental rules of the MATLAB® language which ensure that an object passed as an input argument of a function cannot be modified, or that inside a function or a script there is a strict equivalence between a variable name and an object. As a first step, EVARIX™ currently enforces some limitations to the use of handle class objects to ensure the safety of its optimizations:

⚠ Limitation

- assignments between handle class objects are not supported[a].
- passing handle class objects as input parameters of regular functions, or as input parameters of other class methods is not supported.

[a] Except of course for the call to the class constructor.

4.7 Formatted I/Os

EVARIX™ supports a number of functions following the C `printf` format operand syntax. The formatting string is currently passed to the equivalent C function without any modification. This may lead to erroneous outputs when executing the code generated by EVARIX™, when the type inferred for a given variable is more precise than the default type assumed by MATLAB®. You may then have to change the formatting string for the faulty argument.

For instance, in the following snippet:

```
P = 100;
printf('P = %g\n', P);
```

EVARIX™ infers that the type of variable `P` is an integer, whereas MATLAB® assumes that it is a floating point value, and the `\%g` format specifier expects a floating point number. The correct result is obtained by modifying the formatting string:

```
P = 100;
printf('P = %d\n', P);
```

4.8 Annotations

EVARIX™ provides a number of annotations for driving the compilation and ease the optimization of the code. An annotation (also called a *pragma*) is a formatted text in a MATLAB® comment. This section describes the available annotations and their syntax:

- Optimization annotations (Section 4.8.2)
- Parallel annotations (Section 4.8.3)
- Compilation annotations (Section 4.8.4)

[7] This is similar to the notion of *pointer* or *reference* in other languages.

EVARIX™ automatically detects quite some opportunities to optimize and parallelize the generated code while still ensuring its absolute correctness. However, in some situations, determining without ambiguity the correctness of a foreseen optimization may require a highly complex analysis practically beyond any reasonable cost ... or even be formally impossible: in such cases, to strictly guarantee the correctness of the generated code, EVARIX™ chooses not to take any risk.

Therefore, the user is provided with a way of adding specific assertions and directives to help the compiler identify hidden possible optimizations: this is the purpose of the "annotations" documented in this section. Their use is intended to become less and less necessary as the future versions of the compiler will become smarter.

In practice, it is recommended to perform a first compilation without annotations, and to introduce them later on based on the analysis feed-back provided by the compiler.

4.8.1 Annotations general syntax

An annotation is a MATLAB® comment beginning with the keyword `pragma`, followed by the name of EVARIX™ and the annotation content. For historical reasons, `evarix` can be replaced by the name of the compiler, `cold`. Hence, the supported syntax is:

Annotation - **General Syntax**

```
%pragma evarix <annotation string>
```

or

Annotation - **General Syntax**

```
%pragma cold <annotation string>
```

4.8.2 Optimization annotations

Annotation - **Inlining**

```
%pragma evarix inline
```

When the function inline expansion is triggered by using an `--inline-level` greater or equal to 1, forcing the inlining of a specific function can be achieved either by using the `--inline-function` options, or with the `inline` annotation, which must be placed just before the definition of the function:

```
%pragma evarix inline
function b = foo(a)
 b = a + 1;
end
```

However, functions with multiple return statements and recursive functions are never expanded inline.

Use this feature with care. In most cases, inline expansion may accelerate your code, but it may also slow down the compilation process by increasing the code size, and, in some situations, slow down the generated code.

Annotation - **Static parameter**

```
%pragma evarix static parameter N
```

This annotation specifies that the variable `N` is a static parameter during the execution: its value is never modified, and can be propagated forward in the scope in which the variable is defined. For scalar variables, the propagation is interprocedural. The annotation must be placed before the statement defining the unique value of the variable:

```
%pragma evarix static parameter n
n=100
```

Annotation - **Loop invariant**

```
%pragma evarix invariant [in loop(I)]
```

Indicates that the following statement is an invariant for the loop. If the statement is in nested loops, the `in loop` clause is used for specifying the index of the surrounding loop for which the statement is invariant. Note that this annotation is not interprocedural, which means that the loop index must refer to a loop in the same function as the annotation.

```
for ii = 1:n
  for jj = 1:n
%pragma evarix invariant in loop(jj)
      kk = ii + 1;
      ...
  end
end
```

4.8.3 Parallelism annotations

Annotation - **gpu section**

```
%pragma evarix begin on(gpu)
matlab statements
%pragma evarix end
```

Identifies a region of code which must be executed on the gpu. If a statement in the region cannot be executed on gpu, a warning is issued at compile time, and EVARIX™ automatically generates the code to safely execute that statement on the cpu, moving data back to the cpu if necessary.

When the `--enable-parallel-pragmas` option is used, embedded parallel element wise, parallel blocks and parallel for annotated statements inherit the gpu context of the gpu section.

When the `--auto-parallelization-level` option is used, the embedded element wise statement and for loops wich are identified as being parallel inherit the gpu context of the gpu section.

Beware that currently, user function calls are not supported in gpu sections. When such a function call is encountered in a gpu section, the necessary data are relocated to the cpu, and the statement is executed on the cpu. To avoid unecessary data movements and try to execute the function on the gpu, you may consider inlining the function, either with the `inline` annotation and the `--inline-level=1` option or with the `--inline-level=2` option.

Note

Other parallelism annotations are taken into account only when the `--enable-parallel-pragmas` option is used.

Annotation - **Parallel block**

```
%pragma evarix parallel block [onvariable(var-list)]
statements
%pragma evarix end
```

The parallel block annotation is used to indicate that the region of code performs independent element-wise operations on variables of the same sizes. The onvariable clause specifies which input variables must be traversed element-by-element.

In the current version, the operations in the block be must element-wise arithmetic operations, or calls to the MATLAB® `conv2`, `filter2` and `interp2` functions.

In the following example:

```
%pragma evarix parallel block onvariable(lena)
for i=1:Nstep
    lena_denoised = filter2(gaussFilter, lena);
end
%pragma evarix end
```

EVARIX™ generates an outer parallel loop which iterates over the `lena` and `lena_denoised` arrays element by element, calling an element-wise version of the `filter2` function.

Annotation - **Parallel region**

```
%pragma evarix parallel [regionkind] [clauses]
```

The parallel region annotation is a low-level annotation used to force EVARIX™ to generate parallel code for the annotated statement.

Currently, `regionKind` can be either `for` (when annotating a `for` loop) or `element-wise` (when annotating an element-wise statement, e.g. a matrix or vector operation).

The annotation accepts clauses used to control the behavior of the parallelism on this region, or to specify how the variables of the parallel region interact between threads or with the rest of the code. These clauses are described in the next section.

Parallel Region optional clauses

Note

For the clause to be taken into account, there must be no space between the clause name and the opening parenthesis of the clause arguments.

Parallel Region clause - **number of threads**

```
%pragma evarix parallel [regionKind] nthreads(N)
```

This clause sets the number of threads on which the parallel region must be run.

Parallel Region clause - **firstprivate**

```
%pragma evarix parallel [regionKind] firstprivate(comma-separated-list-of-variables)
```

This clause specifies the list of variables for which each thread must have a private copy initialized to the value held by the variable in the parent thread before the execution of the parallel region. For `regionkind = for` only.

Parallel Region clause - **condition**

```
%pragma evarix parallel [regionKind] if(condition)
```

This clause specifies that the parallel execution depends on the result of the evaluation of `condition` just before the region execution. If it evaluats to true, the execution is performed in parallel, otherwise, it remains sequential.

Parallel Region clause - **lastprivate**

```
%pragma evarix parallel [regionKind] lastprivate(comma-separated-list-of-variables)
```

This clause specifies the list of variables for which each thread must have a private copy. After the execution of the parallel region, the variable in the parent thread holds the value of the iteration corresponding to the last value of the index. For `regionkind = for` only.

Parallel Region clause - **private**

```
%pragma evarix parallel [regionKind] private(comma-separated-list-of-variables)
```

This clause specifies the list of variables for which each thread must have a private copy. No value is imported from or exported to the parent thread. For `regionkind = for` only.

Parallel Region clause - **shared**

```
%pragma evarix parallel [regionKind] shared(comma-separated-list-of-variables)
```

This clause specifies the list of variables shared among the parallel threads. For `regionkind = for` only.

Parallel Region clause - **on**

```
%pragma evarix parallel [regionKind] on(gpu)
```

Currently available for `regionkind = elementwise` and `regionkind = block` only. This clause allows to specify that the following element wise statement or the following block can and must be executed on a gpu.

Annotation - **nthreads**

```
%pragma evarix nthreads(N)
```

This clause sets the number of threads to use for the overall application.

Annotation - **Critical section**

```
%pragma evarix critical
matlab statements
%pragma evarix end
```

Identifies a section in a parallel region that must be executed by only one thread at a time.

4.8.4 Driving the compilation process

Annotation - Entry

```
%pragma evarix entry(FUNCNAME, comma-separated-list-of-types)
function [...] = FUNCNAME(arg1, arg2,...)
```

The entry pragma is used to specify both the entry point (function) and the type of its input parameters for a MEX compilation. The syntax for the list of types is provided in Section 4.2.5.

Annotation - Extern

```
%pragma evarix extern        funcname(types)
%pragma evarix extern type = funcname(types)
%pragma evarix extern [types] = funcname(types)
```

The `extern` pragma is used to specify the signatures (the input and output types) of a function named `funcname` and defined in an external library. This annotation must be placed in the external library description file (see Chapter 3).

Annotation - Importlib

```
%pragma evarix importlib(/path/to/libmylib.m)
```

The `importlib` pragma is used to specify the path to an external library description file (see Chapter 3).

Annotation - Ignore section

```
%pragma evarix ignore
matlab statements to be ignored
%pragma evarix end
```

The `ignore` annotation specifies that the enclosed region of code must be ignored by EVARIX™ during the compilation.

Variables defined or modified within this region must not be used after the region.

This feature is particularly useful for graphic operations not supported by EVARIX™. It allows to use the same code with both MATLAB® and EVARIX™ without modifying the code.

Annotation - Load

```
%pragma evarix load('/path/to/file.mat')
```

This annotation must be placed just before a call to the `load` function, when it is not called with an explicit file name:

```
filename = ...;
...
% pragma evarix load('/path/to/file.mat')
load(filename);
```

`file.mat` must define the same variables, with the same scalar types and shapes as in the file which will be actually loaded during the execution of the generated excutable. For arrays, the number of dimensions is taken into account, but not the numerical sizes of the dimensions. This means that the file actually loaded during the execution may define arrays with the same number of dimensions, but with different dimension sizes.

Annotation - **Set stacksize**

```
%pragma evarix stacksize(N)
```

This annotation is used for specifying the size of the internal COLD© run-time stack (see also the `--stack-size` option documentation, in Section 2.5.2).

Annotation - **Set type of variable**

```
%pragma evarix type (var_name, scalar_type)
```

The `type` annotation is used for setting the element type of a variable. The syntax for specifying this element type is described in Section 4.2.5.

This feature is merely useful when an unusual type must be enforced, for example for a complex of sized-integers variable, or when the values of an array are read from a file.

Beware that the annotation does not apply to a given statement but to a whole function or script part. The scalar type of the variable is considered as definitely fixed for the whole scope of the variable, even if the annotation is placed after statements which already assign values of different types to the variable.

4.9 Unsupported features and known issues

4.9.1 Unsupported features

The following MATLAB® features are currently not supported (non exhaustive list):

- cell arrays, sparse arrays, function handlers and anonymous functions, classes;
- arrays of strings, structures, or class objects;
- dynamic references to structure fields are not supported. A dynamic reference is a reference to a structure field which is unknown at compile time. For example:

```
ref = 'myField';
A.(ref) = 5;
disp(A.(ref));
```

  All references to structure fields must be explicit:

```
A.myField = 5;
disp(A.myField);
```

- the following class features:
  - class definition directories;

  - events and enumerations;
  - most class, properties and methods blocks attributes are ignored;
  - inheritance (except from the `handle` class);
  - calling methods without the `dot` notation;
- try/catch: the try block is considered as a normal region of code, and the catch block is ignored;
- spmd: the Spmd block is considered as a normal region of code;
- persistent variables;
- the `eval` function;
- the concatenation of multidimensional arrays;
- calling a script within another script;
- nested functions.

4.9.2 Other known issues

Constants in lhs and rhs of an assignement

The following code,

```
i=[i 2i 3 4; 5 6+i 7 8];
```

where the symbol `i` represents the imaginary constant on the right hand side of the assignment, and a user variable name on the left hand side, cannot be handled by EVARIX™. It results in a code which cannot be compiled by the back-end C++ compiler. This issue arises for all MATLAB® constants and not only `i`. A workaround is to use variable names different from MATLAB® constant names.

Comments in array definitions

The following code,

```
A(:,:,1)=  [0 0 0 0          %my comment
            0 0 0 0
            0 0 0 0];
```

raises a syntax error. A workaround is to place the comment outside of the array definition.

Element wise `|` and `&` in `if` conditional expression

In the expression `if A & B`, `B` is not evaluated if `A` evaluates to false (this is called *short-circuit* evaluation). In this case, the code generated by EVARIX™ evaluates both `A` and `B`. In most cases, it works fine; however, if this is an issue, a workaround is to nest the conditions:

```
if A
   if B
      ...
```

Similarly, in the expression `if A | B`, `B` must not be evaluated if `A` evaluates to true, whereas the code generated by EVARIX™ evaluates both `A` and `B`. Again, if this is an issue, a workaround is:

```
if A
   ...
elseif B
   ...
```

Growing number of dimensions of variable-sized arrays

EVARIX™ currently fails to correctly take into account the number of dimensions of an array when it changes dynamically. In the following example:

```
A(1)=5
A(2)=10
A(5)=20
A(1,5)=30
A(2,8,2)=70
```

`A` is erroneously considered as a 2D-array and not as a 3D-array.

Clear function in command mode

Clear-ing an undefined symbol, such as in:

```
clear A
A = 10
```

raises an error, as `A` is not known before the statement `clear A`.
However, it works fine if `A` has been previously defined, as in:

```
A = 5;
clear A
A = 10
```

or if the `clear` function is invoked as a regular function:

```
clear('A')
A = 10
```

Removing elements from arrays using []

The following kind of code is not supported by EVARIX™:

```
A = 1:20;
A(8:12) = [];
```

Successive initializations with [] and a string

The following code cannot be handled by EVARIX™:

```
A = []
A = 'hello'
```

because in the the first statement `A` is considered as an array of numerical values, and in the second statement as a string (which is a scalar type) and not as an array of characters or array of strings.

Order of function arguments

The order in which the arguments of a function call are evaluated may be different in MATLAB® and in the code generated by EVARIX™, depending on the back-end C++ compiler. This may be an issue only when the arguments have global side effects, for instance by calling intrinsic MATLAB® functions such as random number generators, or by calling user functions modifying the values of global variables. In these rare cases, a workaround is to assign the expressions to temporary variables prior to the call, as in:

```
tmp1 = rand();
tmp2 = rand();
foo(tmp1, tmp2);
```

Part II

Reference Guide

The following chapters describe the operations and functions supported by eVariX™ and the types of the data they take as arguments. They are classified in three categories:

- language fundamentals (Chapter 5)
- mathematics (Chapter 6)
- other functions (Chapter 7)

In the remainder of this document, the term *array* designates either a matrix or a ND-array, and *scalar* stands for a single value. **Without futher precision, all vectors, matrices and arrays contain numeric data: boolean, integer, real or complex numbers.** Specific cases implying only a subset of these types are detailed.

Since the number of dimensions of a variable is computed at compile time and cannot change during the executionof the generated code[8], the distinction between vector, matrix and arrays is important.

[8] But the size of a dimension may become 1.

CHAPTER 5
LANGUAGE FUNDAMENTALS

5.1 Matrices and arrays

5.1.1 Create arrays

See also: true, false, rand.

➢ **Function**: `zeros`

Create a matrix made of zeros.

Syntax

```
(1)    a = zeros
(2)    A = zeros(sz1, .., szN)
(3)    A = zeros(n)
(4)    A = zeros(V)
```

Description

- `(1)` returns the scalar 0.
- `(2)` creates an `sz1`-by-...-by-`szN` array of 0. The `szi` parameters are integers.
- `(3)` creates an `n`-by-`n` matrix of 0. `n` is an integer.
- `(4)` `V` is a vector of integers defining the size of `A`. This version is supported only if the number of elements of `V` is explicit.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

➢ **Function**: `ones`

Create a matrix made of ones.

Syntax

```
(1)    a = ones
(2)    A = ones(sz1, .., szN)
(3)    A = ones(n)
(4)    A = ones(V)
```

Description

- (1) returns the scalar 1.
- (2) creates an sz1-by-...-by-szN array of 1. The szi parameters are integers.
- (3) creates an n-by-n matrix of 1. n is an integer.
- (4) V is a vector of integers defining the size of A. This version is supported only if the number of elements of V is explicit.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

➢ **Function**: eye

Identity matrix.

Syntax

```
(1)    I = eye
(2)    I = eye(m,n)
(3)    I = eye(m)
(4)    I = eye(V)
```

Description

- (1) returns the scalar 1.
- (2) creates an identity matrix I of integers of size m-by-n. m and n are integers.
- (3) creates an n-by-n identity matrix of integers. n is an integer.
- (4) V is a vector of 2 integers defining the size of the identity matrix of integers I.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

➢ **Function**: diag

Diagonal matrix and diagonals of matrix.

Syntax

```
(1)    D = diag(V)
(2)    D = diag(V,k)
(3)    v = diag(A)
(4)    v = diag(A,k)
```

Description

- (1) returns a diagonal matrix with elements of vector V on the main diagonal.
- (2) places the elements of V on the k-th diagonal. k = 0, represents the main diagonal, k > 0 is above the main diagonal, k < 0 is below the main diagonal. Other elements of D are set to zero.
- (3) returns a vector containing the values of the main diagonal of the matrix A.
- (4) returns a vector containing the values of the k-th diagonal of the matrix A.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

➢ **Function**: repmat

Replicate and tile array.

Syntax

```
(1)    M = repmat(a,n)
(2)    M = repmat(a,m,n)
(3)    M = repmat(A,n)
(4)    M = repmat(A,m,n)
```

Description

- (1) replicates the scalar a to create an n-by-n matrix. n is an integer.
- (2) replicates the scalar a to create an m-by-n matrix. m and n are integers.
- (3) replicates the matrix A to create an n*size(A,1)-by-n*size(A,2) matrix. n is an integer.
- (4) replicates the matrix A to create an m*size(A,1)-by-n*size(A,2) matrix. m and n are integers.

⚠ Limitation

- Only scalars and matrices are supported as first parameter.
- This function only creates matrices. Versions of *repmat* with more than 3 parameters are not supported.

5.1.2 Create grids

➢ **Function**: linspace

Generate a linearly spaced vector.

Syntax

```
(1)    V = linspace(x,y)
(2)    V = linspace(x,y,n)
```

Description

- (1) returns a row vector of 100 evenly spaced doubles between real scalars x and y (included).
- (2) returns a row vector of n evenly spaced doubles between real scalars x and y (included).

Gpu

- This function can be used in gpu sections.

➢ **Function**: `meshgrid`

2-D and 3-D grids.

Syntax

```
(1)    [A, B]    = meshgrid(X,Y)
(2)    [A, B, C] = meshgrid(X,Y,Z)
```

Description

- `(1)` returns 2-D grid coordinates based on the coordinates contained in the vectors `X` and `Y`.
- `(2)` returns 3-D grid coordinates based on the coordinates contained in the vectors `X`, `Y` and `Z`.

⚠ Limitation

`meshgrid` can only take vectors as arguments.

Gpu

- This function can be used in gpu sections.

➢ **Function**: `ndgrid`

Rectangular grid in 2-D and 3-D spaces.

Syntax

```
(1)    [A, B]    = ndgrid(X,Y)
(2)    [A, B, C] = ndgrid(X,Y,Z)
```

Description

- `(1)` replicates the grid vectors (`X`, `Y`) to produce a full grid.
- `(2)` replicates the grid vectors (`X`, `Y`, `Z`) to produce a full grid.

⚠ Limitation

- `ndgrid` can only take vectors as arguments.
- Only the versions of `ndgrid` with 2 and 3 parameters are supported.

5.1.3 Size and Shape

➢ **Function**: `length`

Length of largest array dimension.

Syntax

```
(1)    l = length(X)
(2)    l = length(s)
```

Description

- `(1)` returns the largest array dimension of `X` (1 if `X` is a scalar).
- `(2)` returns the length of string `s`.

➢ **Function**: `size`

Array size.

Syntax

```
(1)    S              = size(A)
(2)    s              = size(A,dim)
(3)    [m,n]          = size(A)
(4)    [sz1, .., sz2] = size(A)
```

Description

- `(1)` returns a row vector containing the sizes of each dimension of `A`. For a scalar, returns `[1,1]`.
- `(2)` returns the size of the `dim`-th dimension. `dim` is an integer .
- `(3)` and `(4)` return the sizes of the scalar or array `A`.

➢ **Function**: `ndims`

Number of array dimensions.

Syntax

```
(1)    x = ndims(A)
```

Description

- `(1)` returns the number of dimensions of array `A`: `x` is 2 if `A` is a scalar, vector or matrix, 3 if `A` is a 3D array...

➢ **Function**: `numel`

Number of array elements.

Syntax

```
(1)    x = numel(A)
```

Description

- `(1)` returns the number of elements of scalar or array `A`.

5.1.4 Reshape and rearrange

➢ **Function**: `circshift`

Shift array circularly.

Syntax

```
(1)    B = circshift(A, s)
(2)    B = circshift(A, V)
```

Description

- `(1)` circularly shifts the array `A` by `s` positions along the first dimension whose size is not equal to 1. `s` is an integer.
- `(2)` circularly shifts the i-th dimension of the array `A` by the i-th value in the vector of integers `V`.

In both cases :

- if the shift factor is positive, the values of A are shifted down (or right);
- if the shiftsize is negative, the values of A are shifted up (or left);
- if 0, the values are not shifted.

➢ **Function**: `flipdim`

Flip array dimensions.

Syntax

```
(1)    B = flipdim(A,dim)
```

Description

- `(1)` The `flipdim` function flips the `A` components along the `dim` dimension.

`dim` must be a positive integer less or equal to the number of dimensions of `A`.

➢ **Function**: `fliplr`

Flip array left to right.

Syntax

```
(1)    B = fliplr(A)
```

Description

- `(1)` flips columns of matrix `A` in the left-to-right direction.

⚠ Limitation

N-dimensional arrays are not supported.

➢ **Function**: `flipud`

Flip array up to down.

Syntax

```
(1)    B = flipud(A)
```

Description

- `(1)` flips rows of matrix `A` in the up-to-down direction.

⚠ Limitation

N-dimensional arrays are not supported.

➢ **Function**: `reshape`

Reshape an array.

Syntax

```
(1)    B = reshape(A, m, n)
(2)    B = reshape(A, l, m, n)
(3)    B = reshape(A, l, m, n, p)
(4)    B = reshape(M, m, [])
(5)    B = reshape(M, [], n)
```

Description

- `(1)` reshapes the array `A` by generating an `m`-by-`n` matrix of the same type of elements. `m` and `n` are integers.
- `(2)` reshapes the array `A` by generating an `l`-by-`m`-by-`n` array of the same type of elements. `l`, `m` and `n` are integers.

- (3) reshapes the array A by generating an l-by-m-by-n-by-p array of the same type of elements. l, m, n and p are integers.
- (4) and (5) reshape the matrix M to m-by-numel(A)/m and numel(A)/n-by-n resp. m and n are integers and must be divisors of numel(A).

➢ **Function**: rot90

Rotate matrix 90 degree.

Syntax

```
(1)    B = rot90(A)
```

Description

- (1) rotates matrix A counterclockwise by 90 degrees.

⚠ Limitation

N-dimensional arrays are not supported.

➢ **Function**: sort

Sort array elements.

Syntax

```
(1)    B = sort(A)
(2)    B = sort(A, dim)
(3)    [B, Index] = sort(A)
(2)    [B, Index] = sort(A, dim)
```

Description

- (1) sorts the elements of array A in ascending order along the first array dimension whose size does not equal 1.
- (2) sorts the elements of array A in ascending order along the dimension dim.
- The (3) and (4) versions also return the Index array of integers of the same size as A describing the arrangement of elements of A into B.

➢ **Function**: squeeze

Remove singleton dimensions of a 3D-array.

Syntax

```
(1)    B = squeeze(A)
```

Description

- `(1)` removes the singleton dimensions of the 3D-array `A` and returns a matrix `B`.

⚠ Limitation

- `squeeze` can only take a 3D-array as argument.
- this function works only if the singleton dimension can be found at compile time, meaning that the singleton dimension is explicit.

➢ **Function**: `transpose`

Transpose matrix.

Syntax

```
(1)    B = transpose(A)
```

Description

- `(1)` `B` is the transposed matrix of `A`.

Gpu

- This function can be used in gpu sections.

See also the ' operator.

➢ **Function**: `ctranspose`

Complex conjugate transpose matrix.

Syntax

```
(1)    B = ctranspose(A)
```

Description

- `(1)` `B` is the complex conjugate transposed matrix of `A`.

Gpu

- This function can be used in gpu sections.

See also the .' operator.

5.1.5 Misc

➢ **Function**: `ind2sub`

Subscripts from linear index.

Syntax

```
(1)    [I,J]   = ind2sub(s, INDEX)
(2)    [I,J,K] = ind2sub(s, INDEX)
```

Description

- `(1)` and `(2)` compute the equivalent subscript values of linear indexes contained in `INDEX` matrix, according to the base `s`. `s` is a vector containing the size of a matrice or 3D array. `INDEX` is a matrix containing the linear indexes to be converted. `I`, `J` and `K` are matrices of integers of same dimensions than `INDEX`.

See also the .' operator.

5.2 Structures

➢ **Function**: `struct`

Create structure.

Syntax

```
(1)    B = struct('fieldName1', fieldValue1, .., 'fieldNamen', fieldValuen)
```

Description

- `(1)` `B` is a structure containing the `fieldNamei` fields, having the `fieldValuei` value. The values must be of supported types.

⚠ Limitation

Arrays of structures are not supported.

➢ **Function**: `isstruct`

Determine whether input is structure.

Syntax

```
(1)    B = isstruct(A);
```

Description

- `(1)` `A` is a variable of supported type and `B` is `true` if `A` is a structure, `false` otherwise.

➢ **Function**: `isfield`

Determine if structure has field.

Syntax

```
(1)    B = isfield(A, str);
```

Description

- `(1)` `A` is a structure and `str` a string. `B` is `true` if `A` contains a field named `str`, `false` otherwise.

➢ **Function**: `rmfield`

Remove field in structure.

Syntax

```
(1)    B = rmfield(A, str);
```

Description

- `(1)` `A` is a structure and `str` a string. `B` is a copy of `A` without the field `str`.

➢ **Function**: `getfield`

Get field in structure.

Syntax

```
(1)    field = getfield(A, str);
```

Description

- `(1)` `field` is the field named `str` (string) of the structure `A`.

⚠ Limitation

- `str` must be an explicit string;
- prefer the use of `A.str`.

5.3 Operators and elementary operations

5.3.1 Arithmetic Operations

See 6.1.1.

5.3.2 Relational Operations

All usual relational operators and corresponding functions are supported:

```
%      OP    Function
(1)    <     lt(a,b)  % less than
(2)    <=    le(a,b)  % less or equal to
(3)    >=    ge(a,b)  % greater or equal to
(4)    >     gt(a,b)  % greater than
(5)    ==    eq(a,b)  % equal to
(6)    ~=    ne(a,b)  % not equal to
```

- (1), (2), (3), (4) are defined between scalars and arrays, (5) and (6) are also defined between strings:

  Syntax

  ```
  z = x OP y % with OP one of <, <=, >=, >, ==, ~=
  C = x OP B % with OP one of <, <=, >=, >, ==, ~=
  C = A OP y % with OP one of <, <=, >=, >, ==, ~=
  C = A OP B % with OP one of <, <=, >=, >, ==, ~=
  b = str1 OP str2 % with OP one of ==, ~=
  ```

  Description

  - x and y are scalars.
  - A and B are arrays with the same size.
  - C is a boolean array with the same size as A and B.
  - str1 and str2 are strings.

These operators are available in gpu sections and gpu parallel blocks.

5.3.3 Logical Operations

➢ **Function**: |

Logical OR.

Syntax

```
(1)    z = x | y
(2)    C = A | y
(3)    C = x | B
(4)    C = A | B
(5)    C = or(x,y)
(6)    C = or(A,y)
(7)    C = or(x,B)
(8)    C = or(A,B)
(9)    C = or(A,B,direction)
```

Description

- (1) and (5) logical OR between **real** scalars x and y.
- (2), (3), (6), (7) logical OR between **real** scalars x and y and **real** matrices A and B.
- (4) and (8) logical OR between **real** arrays A and B with the same size.
- z is a boolean, C is a boolean array with same size as A and B.
- For logical operators, both operands are always evaluated.

➢ **Function**: ||

Short-circuit OR.

Syntax

```
(1)    z = x || y
```

Description

- (1) short-circuit OR between **reals** x and y.
- Short-circuit OR means that if the first operand is true, the second operand is not evaluated.
- Short-circuit can only be performed between scalar logical values.

➢ **Function**: &

Logical AND.

Syntax

```
(1)    z = x & y
(2)    C = A & y
(3)    C = x & B
(4)    C = A & B
(5)    C = and(x,y)
(6)    C = and(A,y)
(7)    C = and(x,B)
(8)    C = and(A,B)
(9)    C = and(A,B,direction)
```

Description

- (1) and (5) logical AND between **real** scalars x and y.
- (2), (3), (6) and (7) logical AND between **real** scalars x and y and **real** matrices A and B.
- (4) and (8) logical AND between **real** arrays A and B of same size.
- z is a boolean, C is a boolean array with the same size as A and B.
- For logical operators, both operands are always evaluated.

➢ **Function**: `&&`

Short-circuit AND.

Syntax

```
(1)    z = x && y
```

Description

- `(1)` short-circuit AND between **reals** `x` and `y`.
- Short-circuit AND means that if the first operand is false, the second operand is not evaluated.
- Short-circuit can only be performed between scalar logical values.

➢ **Function**: `all`

Determine if all array elements are non-zero or true.

Syntax

```
(1)    z = all(x)
(2)    z = all(V)
(3)    B = all(A)
(3)    B = all(A, dim)
```

Description

- `(1)` returns true if **real** scalar `x` is non-zero (or true), false otherwise.
- `(2)` returns true if all elements of **real** vector `V` are non-zero (or true), false otherwise.
- `(3)` acts along the first dimension of **real** array `A` whose size is not 1, and returns an array of boolean values. The size of this dimension becomes 1 in the boolean array `B`.
- `(4)` acts along the `dim` dimension of **real** array `A` and returns an array of boolean values. The size of dimension `dim` becomes 1 in the boolean array `B`.

➢ **Function**: `any`

Determine if any array element is non-zero or true.

Syntax

```
(1)    z = any(x)
(2)    z = any(V)
(3)    B = any(A)
(3)    B = any(A, dim)
```

Description

- `(1)` returns true if scalar `x` is non-zero (or true), false otherwise.
- `(2)` returns true if vector `V` has any non-zero (or true) element, false otherwise.

- (3) acts along the first dimension of array `A` whose size is not 1 and returns an array of boolean values. The size of this dimension becomes 1 in the boolean array `B`.
- (4) acts along the `dim` dimension of array `A` and returns an array of boolean values. The size of dimension `dim` becomes 1 in the boolean array `B`.

➢ **Function**: `islogical`

Determine if input is logical.

Syntax

```
(1)    z = islogical(x)
(2)    B = islogical(A)
```

Description

- (1) returns true if scalar `x` is logical (boolean).
- (2) returns true if array `A` is an array of boolean values.

➢ **Function**: `false`

Create a matrix of all false.

Syntax

```
(1)    a = false
(2)    A = false(sz1, .., szN)
(3)    A = false(n)
(4)    A = false(V)
```

Description

- (1) returns the boolean `false`.
- (2) creates an `sz1`-by-...-by-`szN` array of `false` values. The `szi` parameters are integers.
- (3) creates an `n`-by-`n` matrix of `false` values. The `n` parameter is an integer.
- (4) `V` is a vector of integers defining the sizes of the dimensions of `A`. This version is supported only if the number of elements of `V` is explicit.

➢ **Function**: `find`

Find indices of nonzero elements in array.

Syntax

```
(1)    V = find(A)
(2)    V = find(A, nmax)
(3)    V = find(A, nmax, direction)
```

Description

- (1) returns a vector of integers containing the indices of non-zero values of the array `A`.

- (2) returns a vector of integers of maximal size `nmax` containing the indices of the first `nmax` non-zero values of the array `A`.
- (3) same as (2) with a string `direction` equals to `'last'` to find the last `nmax` nonzero values, `'first'` (default) to find the first `nmax` non-zero values.

➢ **Function**: `~`

Logical not.

Syntax

```
(1)    z = ~x
(2)    B = ~A
(3)    z = not(x)
(4)    B = not(A)
```

Description

- (1) and (3): logical negation of scalar value `x`.
- (2) and (4): element-wise logical negation of values of array `A`.

➢ **Function**: `true`

Create a matrix of all true.

Syntax

```
(1)    a = true
(2)    A = true(sz1, .., szN)
(3)    A = true(n)
(4)    A = true(V)
```

Description

- (1) returns the boolean `true`.
- (2) create an `sz1`-by-...-by-`szN` array of `true` values. The `szi` parameters are integers.
- (3) create an `n`-by-`n` matrix of `true` values. The `n` parameter is an integer.
- (4) `V` is a vector of integers defining the sizes of the dimensions of `A`. This version is supported only if the number of elements of `V` is explicit.

➢ **Function**: `xor`

Logical exclusive or.

Syntax

```
(1)    z = xor(x,y)
(2)    C = xor(x,A)
(3)    C = xor(A,x)
(4)    C = xor(A,B)
```

Description

- (1) logical exclusive or between **real** values x and y.
- (2) and 3 element-wise exclusive or between **real** value x and **real** array A.
- (4) element-wise exclusive or between **real** arrays A and B with the same size.
- C is an array of boolean with the same size as A and B.

5.3.4 Bit-Wise Operations

➢ **Function**: bitand

Bit-wise AND.

Syntax

```
(1)    z = bitand(x,y)
(2)    C = bitand(x,A)
(3)    C = bitand(A,x)
(4)    C = bitand(A,B)
```

Description

- (1) bit-wise AND between **real** values x and y.
- (2) and 3 element-wise bit-wise AND operation between **real** value x and **real** array A.
- (4) element-wise bit-wise AND between **real** arrays A and B with the same size.
- C is an array of boolean with the same size as A and B.

➢ **Function**: bitor

Bit-wise OR.

Syntax

```
(1)    z = bitor(x,y)
(2)    C = bitor(x,A)
(3)    C = bitor(A,x)
(4)    C = bitor(A,B)
```

Description

- (1) bit-wise OR between **real** values x and y.
- (2) and 3 element-wise bit-wise OR operation between **real** value x and **real** array A.
- (4) element-wise bit-wise OR operation between **real** arrays A and B with the same size.
- C is an array of boolean with the same size as A and B.

➢ **Function**: bitxor

Bit-wise XOR.

Syntax

```
(1)    z = bitxor(x,y)
(2)    C = bitxor(x,A)
(3)    C = bitxor(A,x)
(4)    C = bitxor(A,B)
```

Description

- (1) bit-wise XOR between **real** values x and y.
- (2) and 3 element-wise bit-wise XOR operation between **real** value x and **real** array A.
- (4) element-wise bit-wise XOR operation between **real** arrays A and B with the same size.
- C is an array of boolean with the same size as A and B.

5.3.5 Set Operations

➢ **Function**: unique

Unique values in array.

Syntax

```
(1)    A = unique(B)
```

Description

- (1) returns the same data as in B, but with no repetition and in sorted order. B is an array of numeric values and A is a column vector of same type of elements.

5.4 Data types

5.4.1 Numeric types

Basic numeric types

eVariX™ offers a limited support for the following numeric types:

int8	uint8	single
int16	uint16	
int32	uint32	
int64	uint64	

For all this types, element-wise operations are supported, along with some specific functions (see the documentation of each function).

To convert a variable into one of these types, use the following syntax (int8 can be replaced by any type name defined in the previous table):

```
(1)    z = int8(x)
(2)    B = int8(A)
```

- (1) converts **real** x into int8.
- (2) converts the values in **real** array A into int8.

Extreme values

➢ **Function**: `realmax`

Largest positive floating-point number.

Syntax

```
(1)    m = realmax
```

Description

- `(1)` returns the largest positive floating-point number in IEEE double precision.

➢ **Function**: `realmin`

Smallest positive normalized floating-point number.

Syntax

```
(1)    m = realmin
```

Description

- `(1)` returns the smallest positive normalized floating-point number in IEEE double precision.

➢ **Function**: `intmax`

Largest integer.

Syntax

```
(1)    m = intmax
```

Description

- `(1)` returns the largest value that can be represented with a 32-bit integer.

➢ **Function**: `intmin`

Smallest integer.

Syntax

```
(1)    m = intmin
```

Description

- `(1)` returns the smallest value that can be represented with a 32-bit integer.

5.4.2 Strings

➢ **Function**: `blanks`

Create a string of blanks.

Syntax

```
(1)    res = blanks(n)
```

Description

- `(1)` returns a string containing `n` blanks. `n` is an integer.

➢ **Function**: `ischar`

Determine if variable is a string.

Syntax

```
(1)    b = ischar(A)
```

Description

- `(1)` returns true if `A` is a string, false otherwise.

➢ **Function**: `isletter`

Determine if characters are letters.

Syntax

```
(1)    res = isletter(str)
```

Description

- `(1)` returns a boolean vector with the same size as the string `str` containing true where elements of `str` are digit characters and false otherwise.

➢ **Function**: `sprintf`

Format data into string.

Syntax

```
(1)    str = sprintf(format, . . . )
```

Description

- `(1)` returns a string formatted according to the `format` string parameter. Other parameters must be scalar values or string.

⚠ Warning

Concerning `sprintf`, EVARIX™ may infer for a parameter a type which is more precise than the default type assumed by MATLAB®. In this case, the format specified in the formatting string may not be adapted anymore, and this may raise warnings or errors during the compilation of the C++ file generated by EVARIX™. To solve this issue, you may have to change the formatting string for the faulty argument.

➢ **Function**: `str2num`

Convert string to integer.

Syntax

```
(1)    v = str2num(str)
```

Description

- `(1)` converts the `str` string to an integer value.

⚠ Limitation

- `str2num` can only process one value. `str2num('1 2 3 4')` is not supported.
- `str2num` can only process integer values. See str2double for real values.

➢ **Function**: `str2double`

Convert string to double.

Syntax

```
(1)    v = str2double(str)
```

Description

- `(1)` converts the `str` string to a real double value.

⚠ Limitation

- `str2double` can only process one value. `str2double('1 2 3 4')` is not supported.

➢ **Function**: `strcmp`

Compare strings.

Syntax

```
(1)    b = strcmp(str1, str2)
```

Description

- (1) returns true if strings `str1` and `str2` are equal.

➢ **Function**: `strcmpi`

Compare strings (case-insensitive).

Syntax

```
(1)    b = strcmpi(str1, str2)
```

Description

- (1) returns true if strings `str1` and `str2` are equal, ignoring any difference in letter case.

➢ **Function**: `strfind`

Find one string within another.

Syntax

```
(1)    V = strfind(str, pattern)
```

Description

- (1) returns a vector of integers containing the indices of the starting index of each occurrence of string `pattern` in string `str`.
- If `pattern` is not found in `str`, `V` is empty.

5.4.3 Date and time

➢ **Function**: `clock`

Current date and time as date vector.

Syntax

```
(1)    c = clock
```

Description

- (1) returns a vector of 6 reals containing the current date and time, following the format: `[year month day hour minute second]`.

➢ **Function**: `date`

Current date string.

Syntax

```
(1)    s = date
```

Description

- (1) returns a string representing the current date.

➢ **Function**: `datenum`

Convert date vectors into serial date numbers.

Syntax

```
(1)    n = datenum(V)
(2)    N = datenum(A)
(3)    N = datenum(Y,M,D)
(4)    N = datenum(Y,M,D,H,MI,S)
```

Description

- (1) converts the date vector `V` (Y,M,D or Y,M,D,H,M,S) into a serial date number (real scalar).
- (2) converts the real matrix `A` in which each line defines a date vector into a vector of serial date numbers with the same number of lines as `A`.
- (3) and (4) convert values into serial date numbers, with:
  - `Y` a scalar or a vector specifying years. Type is integer.
  - `M` a scalar or a vector specifying months. Type is integer.
  - `D` a scalar or a vector specifying days. Type is integer.
  - `H` a scalar or a vector specifying hours. Type is integer.
  - `MI` a scalar or a vector specifying minutes. Type is integer.
  - `S` a scalar or a vector specifying seconds. Type is double.
- for all these parameters, vectors must have the same sizes.

➢ **Function**: `datevec`

Convert serial date number into date vector.

Syntax

```
(1)    A = datevec(DT)
(2)    [Y,M,D,H,I,S] = datevec(DT)
```

Description

- (1) converts a serial date number (defined by datenum) `DT` to a date vector `A` whose format is [year, month, day, hour, minute, second]. If `DT` is a vector of `m` serial date numbers, then `A` is an`m`-by-6 array made of `m` lines of date vectors.
- (2) returns the components of the input date number as a tuple.
  - `Y` is an integer scalar representing the year if `DT` is scalar, an integer vector if `DT` is a vector.
  - `M` is an integer scalar representing the months if `DT` is scalar, an integer vector if `DT` is a vector.
  - `D` is an integer scalar representing the day if `DT` is scalar, an integer vector if `DT` is a vector.
  - `H` is an integer scalar representing the hour if `DT` is scalar, an integer vector if `DT` is a vector.
  - `I` is an integer scalar representing the minute if `DT` is scalar, an integer vector if `DT` is a vector.
  - `S` is an real scalar representing the seconds if `DT` is scalar, a real vector if `DT` is a vector.

➢ **Function**: `eomday`

Last day of month.

Syntax

```
(1)    e = eomday(y, m)
(2)    E = eomday(Y, M)
```

Description

- `(1)` returns an integer representing the last of day of the `m` month (integer) of year `y` (integer).
- `(2)` returns an array of integers of same size as the integer arrays `Y` and `M`.

➢ **Function**: `etime`

Elapsed time.

Syntax

```
(1)    e = etime(t2,t1)
(2)    e = etime(T2,T1)
```

Description

- `(1)` returns the elapsed time in seconds (real value) between the date-time vector `t2` and `t1`.
  - `t2` is the ending time. It is a vector of 6 elements.
  - `t1` is the beginning time. It is a vector of 6 elements.
- `(2)` returns a vector containing the elapsed time between date-time vectors stored as lines of matrices `T1` and `T2`: `E(i) = etime(T2(i,:), T1(i,:))`.
  - If the size of date-time vectors is 6, the format is [Year Month Day Hour Minute Second].

➢ **Function**: `now`

Current date and time as a serial date number.

Syntax

```
(1)    t = now()
```

Description

- `(1)` returns a real representing the current date and time as a serial date number.

➢ **Function**: `weekday`

Day of the week as a serial date number.

Syntax

```
(1)    [A,B] = weekday(C)
(2)    [A,B] = weekday(C,'option')
```

Description

- `(1)` returns the day of the week (both in number and string format) from a date given as a serial date number:
  - `C` is the date passed as a serial number. it can be a scalar, or an array.
  - `A` is the numeric representation of the weekday. It is an integer if `C` is scalar, or an array of the same size as `C`.
  - `B` is the string representation of the weekday, of the same size as `C`.
- `(2)` `option` can be either `'long'` for long weekday names (Monday, Tuesday, Wednesday, etc.) or `'short'` for short weekday names (Mon, Tue, Wed, etc.).

5.4.4 Misc

➢ **Function**: `disp`

Display variable.

Syntax

```
(1)    disp(x)
(2)    disp(A)
(3)    disp(str)
```

Description

- `(1)` displays value of scalar `x`.
- `(2)` displays array `A`.
- `(3)` displays string `str`.

➢ **Function**: `error`

Exit with error and print message.

Syntax

```
(1)    error(str)
```

Description

- `(1)` exits with error and displays the string `str`, unless the string is empty.

➢ **Function**: `exit`

Exit with error code.

Syntax

```
(1)    exit(code)
```

Description

- `(1)` exits with error code `code` (integer).

⚠ Warning

If this function is called in a MEX file, the MATLAB® environment is also exited.

➢ **Function**: `nargin`

Number of function input arguments

Syntax

```
(1)    nargin
(2)    nargin('foo')
```

Description

- `(1)` returns the number of the function actual input parameters in the current call.
- `(2)` returns the expected number of input parameters for function `foo`.

⚠ Limitation

`nargin('foo')` is only supported when `foo` is a user function which is actually used in the code being processed.

➢ **Function**: `nargout`

Number of function output arguments

Syntax

```
(1)    nargout
(2)    nargout('foo')
```

Description

- `(1)` returns the number of the function actual output parameters in the current call.
- `(2)` returns the expected number of output parameters for function `foo`.

⚠ Limitation

`nargout('foo')` is only supported when `foo` is a user function which is actually used in the code being processed.

➢ **Function**: `nargchk`

Check number of function arguments (deprecated)

Syntax

```
(1)    str = nargchk(minVal, maxVal, nargVal)
```

Description

- `(1)` returns a string bearing an error message if `nargVal` is strictly smaller than `minVal` or strictly greater than `maxVal`. Otherwise, returns an empty string. `foo`.

➢ **Function**: `narginchk`

Check number of function input arguments

Syntax

```
(1)    narginchk(minVal, maxVal)
```

Description

- `(1)` throws an error message if `nargin` is strictly smaller than `minVal` or strictly greater than `maxVal`.

➢ **Function**: `nargoutchk`

Check number of function output arguments

Syntax

```
(1)    nargoutchk(minVal, maxVal)
```

Description

- `(1)` throws an error message if `nargout` is strictly smaller than `minVal` or strictly greater than `maxVal`.

➢ **Function**: `format`

Set display format of real values.

Syntax

```
(1)    format(style)
```

Description

- `(1)` changes the display format of real values. `style` is a string.
- supported formats are:
  - `'short'` (default): short, fixed-decimal format with 4 digits after decimal point.
  - `'long'`: long, fixed-decimal format with 15 digits after decimal point.

➢ **Function**: `getenv`

Get value of environment variable.

Syntax

```
(1)    val = getenv(key)
```

Description

- `(1)` gets the value (string) of the environment variable designated by the string `key`.

➢ **Function**: `quit`

Terminate computation.

Syntax

```
(1)    quit
```

Description

- `(1)` terminates the running program.

⚠ Warning

If this function is called in a MEX file, the MATLAB® environment is also exited.

➢ **Function**: `warning`

Display a warning message.

Syntax

```
(1)    warning(msg)
```

Description

- `(1)` displays the warning message `msg` (string).

CHAPTER 6
MATHEMATICS

6.1 Elementary mathematics

6.1.1 Arithmetic

➢ **Function**: +

Addition.

Syntax

```
(1)    z = x + y
(2)    C = x + A
(3)    C = A + x
(4)    C = A + B
```

Description

- (1) addition of scalars **x** and **y**.
- (2) and (3) element-wise addition between scalar `x` and array `A`.
- (3) element-wise addition between arrays `A` and `B`. `A` and `B` have the same size.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator. Support for complex on gpu.

➢ **Function**: -

Subtraction.

Syntax

```
(1)    z = -x
(2)    C = -A
(3)    z = x - y
(4)    C = A - y
(5)    C = x - B
(6)    C = A - B
```

Description

- (1) negative of scalar x.
- (2) element-wise negative of elements of array A.
- (3) subtraction of scalars **x** and **y**.
- (4) and (3) element-wise subtraction between scalar x and array A.
- (5) element-wise subtraction between arrays A and B. A and B have the same size.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator. Support for complex on gpu.

➢ **Function**: *

Multiplication of scalars or arrays.

Syntax

```
(1)      z = x * y
(2)      C = A * x
(3)      C = x * A
(4)      z = X .* Y
(5)      C = A * B
```

Description

- (1) multiplication between scalars x and y.
- (2) and (3) element-wise multiplication between scalar x and array A.
- (4) scalar product between a row vector **X** and a column vector **Y** with the same number of elements. z is a scalar.
- (5) matrix multiplication between **matrices** A and B, with `size(A,2) == size(B,1)`.

➢ **Function**: .*

Element-wise multiplication.

Syntax

```
(1)    z = x .* y
(2)    C = x .* A
(3)    C = A .* x
(4)    C = A .* B
```

Description

- (1) multiplication between scalars x and y.
- (2) and (3) element-wise multiplication between scalar x and array A.
- (4) element-wise multiplication between arrays A and B of same size.

Gpu

- This function can be used in gpu sections, gpu parallel blocks and parallel blocks on gpu : it is available as scalar and element-wise operator. Support complex on gpu.

➢ **Function**: /

Division by scalar values.

Syntax

```
(1)    z = x / y
(2)    B = A / y
```

Description

- (1) division between scalars x and y.
- (2) element-wise division between array A and scalar y.

➢ **Function**: ./

Element-wise division.

Syntax

```
(1)    z = x ./ y
(2)    C = x ./ A
(3)    C = A ./ x
(4)    C = A ./ B
```

Description

- (1) division between scalars x and y.
- (2) and (3) element-wise division between scalar x and array A.
- (4) element-wise division between arrays A and B of same size.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator. Support complex on gpu.

➢ **Function**: \

Left division.

Syntax

```
(1)    z = x \ y
(2)    C = A \ B
```

Description

- (1) left division between scalars x and y: x\y = y/x.
- (2) left division between the square matrix A and the matrix B with the same number of lines as A. C is a matrix of the same size as B such that A*C=B.

➢ **Function**: .\

Left division.

Syntax

```
(1)    z = x .\ y
(2)    C = x .\ A
(3)    C = A .\ x
(4)    C = A .\ B
```

Description

- (1) left division between scalars x and y: x .\ y == y ./ x.
- (2) and (3) element-wise left division between scalar x and array A: x .\ A == A ./ x.
- (4) element-wise left division between arrays A and B of same size: A .\ B == B ./ A

➢ **Function**: ^

Power of scalars or arrays of scalars.

Syntax

```
(1)    z = x ^ y
(2)    C = A ^ y
```

Description

- (1) raises the scalar x to the power of the real scalar y.
- (2) raises each element of the square matrix A to the power of the real scalar y.
- by default, if the operands are real values, the type of the output is real. Use the --complex option to get complex values.

➢ **Function**: .^

Element-wise power.

Syntax

```
(1)    z = x .^ y
(2)    C = A .^ y
(3)    C = x .^ B
(4)    C = A .^ B
```

Description

- (1) raises the scalar x to the power of the real scalar y.
- (2) raises each element of the array A to the power of the real scalar y.
- (3) computes the array with the real scalar x raised to the power of the elements of array B. C has the same size as B.
- (4) Raises each element of A to the power of the corresponding element of B. A and B have the same size.
- by default, if the operands are real values, the type of the output is real. Use the --complex option to get complex values.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator. Support complex on gpu.

➢ **Function**: '

Transpose conjugate matrix.

Syntax

```
(1)    B = A'
```

Description

- (1) transpose conjugate of matrix A.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

➢ **Function**: .'

Transpose matrix.

Syntax

```
(1)    B = A.'
```

Description

- (1) transpose of matrix A.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

Products and sums

➢ **Function**: `cumprod`

Cumulative product.

Syntax

```
(1)    B = cumprod(A)
(2)    B = cumprod(A,dim)
```

Description

- `(1)` returns the cumulative product of array `A`.
  - If `A` is a vector, returns a vector containing the cumulative products of the elements of `A`.
  - If `A` is a matrix, returns a matrix containing the cumulative products for each column of `A`.
  - If `A` is an array, `cumprod` acts along the first non-singleton dimension.
- `(2)` returns the cumulative product of array `A` along the dimension `dim`. `dim` is an integer.

Gpu

- This function can be used in gpu sections. Support for complex on gpu.

➢ **Function**: `cumsum`

Cumulative sum.

Syntax

```
(1)    B = cumsum(A)
(2)    B = cumsum(A,orientation)
```

Description

- `(1)` returns the cumulative sum of array `A`.
  - If `A` is a vector, returns a vector containing the cumulative sum of the elements of `A`.
  - If `A` is a matrix, returns a matrix containing the cumulative sums for each column of `A`.
  - If `A` is an array, `cumprod` acts along the first non-singleton dimension.
- `(2)` return the cumulative sum of array `A` along the dimension `dim`. `dim` is an integer.

Gpu

- This function can be used in gpu sections. Support for complex on gpu.

➢ **Function**: `prod`

Product of array elements.

Syntax

```
(1)    B = prod(A)
(2)    B = prod(A,dim)
```

Description

- (1) returns the product of the elements of array A along the first non-singleton dimension.
- (2) returns the product of the elements of array A along the dimension dim. dim is an integer between 1 and ndims(A).
- The size of B depends on the size of A and the dimension along which the product is done. This dimension is set to 1 in the resulting array.
- In some cases, eVariX™ does not have enough information to correctly compute the number of dimensions of B, and a warning is issued.

Gpu

- This function can be used in gpu sections.

➢ **Function**: sum

Sum of array elements.

Syntax

```
(1)    B = sum(A)
(2)    B = sum(A,dim)
```

Description

- (1) returns the sum of the elements of array A along the first non-singleton dimension.
- (2) returns the sum of the elements of array A along the dimension dim. dim is an integer between 1 and ndims(A).
- The size of B depends on the size of A and the dimension along which the sum is done. This dimension is set to 1 in the resulting array.
- In some cases, eVariX™ does not have enough information to correctly compute the number of dimensions of B, and a warning is issued.

Gpu

- This function can be used in gpu sections.

Remainders

➢ **Function**: mod

Remainder after division, modulo operation.

Syntax

```
(1)    z = mod(x,y)
(2)    C = mod(A,x)
(3)    C = mod(x,A)
(4)    C = mod(A,B)
```

Description

- (1) modulo of scalar x by scalar y.
- (2) element-wise modulo of array A by scalar x.
- (3) element-wise modulo of scalar x by array A.

- (3) element-wise modulo of array A by array B. A and B have the same size.
- The result is 0 or has the same sign as the divisor.

➢ **Function**: rem

Remainder after division.

Syntax

```
(1)    z = rem(x,y)
(2)    C = rem(A,x)
(3)    C = rem(x,A)
(4)    C = rem(A,B)
```

Description

- (1) remainder after division of scalar x by scalar y.
- (2) element-wise remainder after division of array A by scalar x.
- (3) element-wise remainder after division of scalar x by array A.
- (4) element-wise remainder after division of array A by array B. A and B have the same size.
- The result is 0 or has the same sign as the dividend.

Rounding

➢ **Function**: ceil

Rounding up.

Syntax

```
(1)    y=ceil(x)
(2)    B=ceil(A)
```

Description

- (1) rounds up the scalar x.
- (2) rounds up the elements of array A.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

➢ **Function**: fix

Rounding towards zero.

Syntax

```
(1)    y = fix(x)
(2)    B = fix(A)
```

Description

- (1) rounds the scalar x towards zero.
- (2) rounds the elements of array A towards zero.

➢ **Function**: floor

Rounding down.

Syntax

```
(1)    y = floor(x)
(2)    B = floor(A)
```

Description

- (1) rounds down the scalar x.
- (2) rounds down the elements of array A.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

➢ **Function**: round

Rounding to the nearest integer.

Syntax

```
(1)    y=round(x)
(2)    B=round(A)
```

Description

- (1) rounds the scalar x to the nearest integer.
- (2) rounds the elements of array A to the nearest integer.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

6.1.2 Trigonometry

All trigonometric functions are element-wise functions. The following table describes the supported functions. All the functions in the table take one parameter, scalar or array, and return a result of same size.

For some of the inverse functions, identified by a *Yes* in the last column, eVariX™ generates by default real output values in case of real input values. To get complex results, use the `--complex` option.

Function	Description	Function	Description	$\mathbb{R}$ or $\mathbb{C}$
Sine				
sin	Sine in radians	asin	Inverse sine in radians	Yes
sind	Sine in degrees	asind	Inverse sine in degrees	Yes
sinh	Hyperbolic sine	asinh	Inverse hyperbolic sine	Yes
Cosine				

cos	Cosine in radians	acos	Inverse cosine in radians	Yes
cosd	Cosine in degrees	acosd	Inverse cosine in degrees	Yes
cosh	Hyperbolic cosine	acosh	Inverse hyperbolic cosine	
Tangent				
tan	Tangent in radians	atan	Inverse tangent in radians	
tand	Tangent in degrees	atand	Inverse tangent in degrees	
tanh	Hyperbolic tangent	atanh	Inverse hyperbolic tangent	Yes
Cosecant				
csc	Cosecant in radians	acsc	Inverse cosecant in radians	Yes
cscd	Cosecant in degrees	acscd	Inverse cosecant in degrees	Yes
csch	Hyperbolic cosecant	acsch	Inverse hyperbolic cosecant	
Secant				
sec	Secant in radians	asec	Inverse secant in radians	Yes
secd	Secant in degrees	asecd	Inverse secant in degrees	Yes
sech	Hyperbolic secant	asech	Inverse hyperbolic secant	Yes
Cotangent				
cot	Cotangent in radians	acot	Inverse cotangent in radians	
cotd	Cotangent in degrees	acotd	Inverse cotangent in degrees	
coth	Hyperbolic cotangent	acoth	Inverse hyperbolic cotangent	Yes

The following functions can be used in gpu sections and gpu parallel blocks, for scalar and element-wise operations: `cos`, `cosh`, `sin`, `sinh`, `tan`, `tanh`,`acos`, `acosh`, `asin`, `asinh`, `atan`, `atanh`.

➢ **Function**: `atan2`

Four-quadrant inverse tangent.

Syntax

```
(1)    z = atan2(x,y)
(2)    C = atan2(A,x)
(3)    C = atan2(x,A)
(4)    C = atan2(A,B)
```

Description

- `(1)` four-quadrant inverse tangent between scalar `x` by scalar `y`.
- `(2)` element-wise four-quadrant inverse tangent of array `A` by scalar `x`.
- `(3)` element-wise four-quadrant inverse tangent of scalar `x` by array `A`.
- `(3)` element-wise four-quadrant inverse tangent of array `A` by array `B`. `A` and `B` have the same size.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: `hypot`

Square root of sum of squares.

Syntax

```
(1)    z = hypot(x,y)
(2)    C = hypot(A,x)
(3)    C = hypot(x,A)
(4)    C = hypot(A,B)
```

Description

- (1) square root of the sum of the squares of scalars x and y.
- (2) and (3) element-wise square root of the sum of the squares of scalar x and array A.
- (3) element-wise square root of the sum of squares of arrays A and B. A and B have the same size.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

6.1.3 Complex numbers

➢ **Function**: abs

Absolute value and complex magnitude.

Syntax

```
(1)    y = abs(x)
(2)    m = abs(z)
(3)    B = abs(A)
```

Description

- (1) returns the absolute value of the real x.
- (2) returns the complex magnitude of the complex z. m is real.
- (3) returns the absolute value or complex magnitude of the elements of the array A. B is an array of real values of the same size as A

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: angle

Phase angle of complex number.

Syntax

```
(1)    y = angle(x)
(2)    B = angle(A)
```

Description

- (1) returns the phase angle of the scalar x. y is a real.
- (2) returns the phase angle of the elements of the array A. B is an array of real values of the same size as A.

➢ **Function**: complex

Create complex array.

Syntax

```
(1)    c = complex(r,i)
(2)    C = angle(R,I)
```

Description

- (1) returns a complex number formed by the real r for the real part and by the real i for the imaginary part.
- (2) returns an array of complex values from the real arrays R and I such as C(:) = complex(R(:),I(:)). R, I and C have the same size.

➢ **Function**: conj

Complex conjugate.

Syntax

```
(1)    y = conj(x)
(2)    B = conj(A)
```

Description

- (1) returns the complex conjugate of scalar x.
- (2) returns the complex conjugates of the elements of array A.

Gpu

- This function can be used on gpu.

➢ **Function**: imag

Imaginary part of complex numbers.

Syntax

```
(1)    y = imag(x)
(2)    B = imag(A)
```

Description

- (1) returns the imaginary part of scalar x.

- (2) returns the imaginary parts of the elements of array A.

➢ **Function**: real

Real part of complex numbers.

Syntax

```
(1)    y = real(x)
(2)    B = real(A)
```

Description

- (1) returns the real part of scalar x.
- (2) returns the real parts of the elements of array A.

➢ **Function**: sign

Sign function

Syntax

```
(1)    y = sign(x)
(2)    B = sign(A)
```

Description

- (1) returns the sign of scalar x.
- (2) returns the sign of the elements of array A.
- The sign of a value x is an integer defined as:
  - 1 if x > 0;
  - 0 if x == 0;
  - -1 if x < 0;
  - x ./ abs(x) if x is complex.

6.1.4 Exponents and Logarithms

➢ **Function**: exp

Exponential

Syntax

```
(1)    y = exp(x)
(2)    B = exp(A)
```

Description

- (1) returns the exponential of scalar x.
- (2) returns the exponentials of the elements of array A.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: `log`

Logarithm.

Syntax

```
(1)    y = log(x)
(2)    B = log(A)
```

Description

- `(1)` returns the logarithm of scalar `x`.
- `(2)` returns the logarithms of the elements of array `A`.
- By default, for real values `log` returns real values (and therefore `nan` if the value is less or equal 0). Use the `--complex` option to return complex numbers.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: `log2`

Base 2 logarithm.

Syntax

```
(1)    y = log2(x)
(2)    B = log2(A)
(3)    [f,e] = log2(x)
(4)    [F,E] = log2(A)
```

Description

- `(1)` returns the base 2 logarithm of scalar `x`.
- `(2)` returns the base 2 logarithms of the elements of array `A`.
- `(3)` returns real `f` and integer `e` such as `x = f .* 2^e`.
- `(4)` returns real array `F` and integer array `E` such as `X = F .* 2^E`.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: `log10`

Base 10 logarithm.

Syntax

```
(1)    y = log10(x)
(2)    B = log10(A)
```

Description

- `(1)` returns the base 10 logarithm of scalar `x`.
- `(2)` returns the base 10 logarithms of the elements of array `A`.
- By default, for real values `log` returns real values (and therefore `nan` if the value is less or equal 0). Use the `--complex` option to return complex numbers.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: `log1p`

Compute `log(1+x)`.

Syntax

```
(1)    y = log1p(x)
(2)    B = log1p(A)
```

Description

- `(1)` returns `log(1+x)` for scalar `x`.
- `(2)` returns `log(1+A)` for array `A`.
- By default, for real values `log` returns real values (and therefore `nan` if the value is less or equal 0). Use the `--complex` option to return complex numbers.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: `pow2`

Base 2 power.

Syntax

```
(1)    y = pow2(x)
(2)    B = pow2(A)
(3)    z = pow2(x,y)
(4)    Z = log2(F,E)
```

Description

- (1) returns 2 raised to the power scalar x.
- (2) returns an array containing 2 raised to the power of the corresponding elements of array A: B(:) = 2 .^ A(:).
- (3) returns a real scalar such that z = x .* 2^e, with x a real and y an integer.
- (4) returns a real array such that Z = F .* 2 .^ E. F is a real array, E an array of integers.

➢ **Function**: nextpow2

Next higher power of 2.

Syntax

```
(1)    p = nextpow2(x)
(2)    B = nextpow2(A)
```

Description

- (1) When x is a scalar, returns the first p such that 2^p >= abs(x).
- (2) Applies nextpow2 element-wise to the array A.

➢ **Function**: reallog

Real logarithm.

Syntax

```
(1)    y = reallog(x)
(2)    B = reallog(A)
```

Description

- (1) returns the logarithm of the non-negative scalar x.
- (2) returns the logarithms of the elements of array A. The elements of A must be non-negative.

➢ **Function**: realpow

Array power for real-only output

```
(1)    z = realpow(x,y)
(2)    C = realpow(A,B)
```

Description

- (1) returns the scalar x to the power of the scalar y.
- (2) raises all element of array A to the power of its corresponding element in array B.
- In both versions, the results must be real.

➢ **Function**: `realsqrt`

Real square-root.

```
(1)    y = realsqrt(x)
(2)    B = realsqrt(A)
```

Description

- `(1)` returns the real square-root of scalar `x`. `x` must be non-negative
- `(2)` returns the real square-root of the elements of array `A`. The elements of `A` must be non-negative.

➢ **Function**: `sqrt`

Square-root.

```
(1)    y = sqrt(x)
(2)    B = sqrt(A)
```

Description

- `(1)` returns the square-root of scalar `x`.
- `(2)` returns the square-roots of the elements of array `A`.
- By default, for real values `sqrt` returns real values (and therefore `nan` if the value is less or equal 0). Use the `--complex` option to return complex numbers.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

6.1.5 Special functions

➢ **Function**: `gamma`

Gamma function.

Syntax

```
(1)    y = gamma(x)
(2)    B = gamma(A)
```

Description

- `(1)` returns the gamma function of the real `x`.
- `(2)` returns the gamma function of the elements of the real array `A`.
- The `gamma` function interpolates the factorial function.

6.1.6 Constant and test matrices

➢ **Function**: `compan`

Companion matrix.

Syntax

```
(1)    C = compan(V)
```

Description

- `(1)` returns the companion matrix of vector `V`. `H` is a matrix of size `(numel(V)-1)`-by-`(numel(V)-1)`.

➢ **Function**: `hankel`

Hankel matrix.

Syntax

```
(1)    H = hankel(V)
(1)    H = hankel(X,Y)
```

Description

- `(1)` returns a square Hankel matrix. `V` is a vector. `H` is a matrix of size `numel(V)`-by-`numel(V)`.
- `(2)` returns a Hankel matrix. `X` and `Y` are vectors. `H` is a matrix of size `numel(X)`-by-`numel(Y)`.

➢ **Function**: `hilb`

Hilbert matrix.

Syntax

```
(1)    H = hilb(n)
```

Description

- `(1)` returns the Hilbert matrix of order `n`. `n` is an integer. `H` is a real matrix of size `n`-by-`n`.

➢ **Function**: `toeplitz`

Toeplitz matrix.

Syntax

```
(1)    T = toeplitz(r)
(2)    T = toeplitz(c,r)
```

Description

- (1) returns a symmetric Toeplitz matrix, with c as its first column and r as its first row.
- (2) returns a non-symmetric Toeplitz matrix.

  If r is a real vector, then r defines the first row of the matrix. If r is a complex vector with a real first element, then r defines the first row and r' defines the first column.

  If the first element of r is complex, the Toeplitz matrix is Hermitian off the main diagonal, which means $T_{i,j} = conj(T_{j,i}) \forall i \neq j$. The elements of the main diagonal are set to r(1).

➢ **Function**: inf

Create a matrix of all infinite values.

Syntax

```
(1)    a = inf                    a = Inf
(2)    A = inf(sz1, .., szN)      A = Inf(sz1, .., szN)
(3)    A = inf(n)                 A = Inf(n)
(4)    A = inf(V)                 A = Inf(V)
```

Description

- (1) returns the scalar inf.
- (2) create a sz1-by-...-by-szN array of inf. The szi parameters are integers.
- (3) create a n-by-n matrix of inf. The n parameter is an integer.
- (4) V is a vector of integers defining the size of A. This version is supported only if the number of elements of V is explicit.

➢ **Function**: invhilb

Inverse of Hilbert matrix.

Syntax

```
(1)    H = invhilb(n)
```

Description

- (1) returns the inverse of Hilbert matrix of order n. n is an integer. H is a real matrix of size n-by-n.

➢ **Function**: isfinite

Determine whether the array elements are finite.

Syntax

```
(1)    y = isfinite(x)
(2)    B = isfinite(A)
```

Description

- (1) returns true if scalar x is finite.
- (2) returns an array of booleans of same size as A containing true where the elements of A are finite.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: isinf

Determine whether the array elements are infinite (inf).

Syntax

```
(1)    y = isinf(x)
(2)    B = isinf(A)
```

Description

- (1) returns true if scalar x is inf or -inf.
- (2) returns an array of booleans of same size as A containing true where the elements of A are inf or -inf.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: isnan

Determine whether the array elements are nan.

Syntax

```
(1)    y = isnan(x)
(2)    B = isnan(A)
```

Description

- (1) returns true if scalar x is nan or -nan.
- (2) returns an array of booleans of same size as A containing true where the elements of A are nan or -nan.

Gpu

- This function can be used in gpu sections and gpu parallel blocks: it is available as scalar and element-wise operator.

➢ **Function**: `isreal`

Determine whether the array is real.

Syntax

```
(1)    a = isreal(A)
```

Description

- `(1)` returns true if `A` is real. `A` can be scalar or array.

➢ **Function**: `nan`

Create a matrix of all nan.

Syntax

```
(1)    a = nan                    a = NaN
(2)    A = nan(sz1, .., szN)      A = NaN(sz1, .., szN)
(3)    A = nan(n)                 A = NaN(n)
(4)    A = nan(V)                 A = NaN(V)
```

Description

- `(1)` returns the scalar `nan`.
- `(2)` create as `sz1`-by-...-by-`szN` array of `nan`. The `szi` parameters are integers.
- `(3)` create as `n`-by-`n` matrix of `nan`. The `n` parameter is an integer.
- `(4)` `V` is a vector of integers defining the size of `A`. This version is supported only if the number of elements of `V` is explicit.

Gpu

- This function can be used in gpu sections and gpu parallel blocks.

➢ **Function**: `pascal`

Pascal matrix.

Syntax

```
(1)    P = pascal(n)
(2)    P = pascal(n,1)
(3)    P = pascal(n,2)
```

Description

- `(1)` returns the Pascal matrix of order `n` (integer): a symmetric positive definite matrix with integer entries taken from Pascal's triangle. `P` is a square matrix of integers of size `n`-by-`n`.
- `(2)` returns the lower triangular Cholesky factor of the Pascal matrix. `P` is a square matrix of integers of size `abs(n)`-by-`abs(n)`.

- (3) returns a transposed and permuted version of pascal(n,1).

➢ **Function**: rosser

Rosser matrix.

Syntax

```
(1)    a = rosser
```

Description

- (1) returns the Rosser matrix. This matrix is an 8-by-8 matrix of integers.

➢ **Function**: vander

Vandermonde matrix.

Syntax

```
(1)    M = vander(V)
```

Description

- (1) returns the Vandermonde matrix containing the powers of the elements of vector V. The resulting matrix M is of size numel(V)-by-numel(V).
- by default, if the vector contains real values, the type of the output is real. Use the --complex option to get complex values.

➢ **Function**: wilkinson

Wilkinson's eigenvalue test matrix.

Syntax

```
(1)    W = wilkinson(n)
```

Description

- (1) returns the Wilkinson matrix of reals of size n-by-n.

6.2 Linear algebra

6.2.1 Linear equations

See also: /, \

➢ **Function**: `inv`

Matrix inverse.

Syntax

```
(1)    Y = inv(X)
```

Description

- `(1)` returns the inverse of the square matrix `X`.

Gpu

- This function can be used in parallel blocks on gpu (local version only).

➢ **Function**: `pinv`

Moore-Penrose pseudoinverse of matrix.

Syntax

```
(1)    B = pinv(A)
(2)    B = pinv(A, tol)
```

Description

- `(1)` returns the pseudoinverse matrix of the matrix `A`. The matrix `B` has same size as `A'`. The singular values less than the default tolerance `max(size(A))*eps(norm(A))` are treated as zero.
- `(2)` returns the pseudoinverse matrix of the matrix `A`. The matrix `B` has same size as `A'`, and overrides the default tolerance with the real `tol`.

6.2.2 Eigenvalues and Singular values

➢ **Function**: `eig`

Eigenvalues and pencils.

Syntax

```
(1)    e     = eig(A)
(2)    [V,D] = eig(A)
```

Description

- `(1)` returns a vector of complex values containing the eigenvalues of the square matrix `A`.
- `(2)` returns a diagonal complex square matrix `D` of eigenvalues, and a complex square matrix V whose columns are the corresponding right eigenvectors, such that `A*V = V*D`.

➢ **Function**: `hess`

Hessenberg form of matrix.

Syntax

```
(1)    [P,H] = hess(A)
```

Description

- `(1)` returns a Hessenberg square matrix `H` and a unitary matrix P such that `A = P*H*P'` and `P'*P = eye(size(A))`. `A` is a square matrix.

➢ **Function**: `schur`

Schur decomposition of matrix.

Syntax

```
(1)    [U,T] = schur(A)
```

Description

- `(1)` returns the Schur square matrix `T` of the square matrix `A`, and a unitary matrix `U` such that `A = U*T*U'` and `U'*U = eye(size(A))`.

➢ **Function**: `svd`

Singular value decomposition.

Syntax

```
(1)    s = svd(X)
(2)    [U,S,V] = svd(X)
(3)    [U,S,V] = svd(X, 'econ')
(4)    [U,S,V] = svd(X, 0)
```

Description

- `(1)` returns a vector containing the singular values of matrix `A`.
- `(2)` performs the singular value decomposition of matrix `A`, such that `A = U*S*V'`, with:
  - `U` a matrix of size `size(A,1)`-by-`size(A,1)`;
  - `S` a diagonal matrix of size equal to `size(A)`;
  - `V` a matrix of size `size(A,2)`-by-`size(A,2)`;
- `(3)` returns an economy size decomposition of the matrix `A`:
  - if `size(A,1) > size(A,2)`, only the first `size(A,2)` columns of `U` are computed and `S` is of size `size(A,2)`-by-`size(A,2)`;
  - if `size(A,1) = size(A,2)`, the computation is same as `svd(A)`;
  - if `size(A,1) < size(A,2)`, only the first `size(A,1)` columns of `V` are computed and `S` is of size `size(A,1)`-by-`size(A,1)`.
- `(4)` returns a different economy-size decomposition: if `size(A,1) > size(A,2)` behaves as `(3)`, else computes `svd(A)`.

6.2.3 Matrix decomposition

➢ **Function**: `chol`

Cholesky factorization.

Syntax

```
(1)    C = chol(A)
```

Description

- `(1)` returns the Cholesky factorization of the square matrix `A`. `B` is a matrix of the same size as `A`.

➢ **Function**: `lu`

LU matrix factorization.

Syntax

```
(1)    Y         = lu(A)
(2)    [L, U]    = lu(A)
(3)    [L, U, P] = lu(A)
```

Description

- `(1)` from the matrix `A`, returns a matrix `Y` of same size as `A` which contains the strictly lower triangular `L` and the upper triangular `U` as submatrices.
- `(2)` from the matrix `A`, returns an upper triangular matrix in `U` and a permuted lower triangular matrix in `L` such that `A = L*U`.
- `(3)` returns the LU decompostion matrices `L` and `U` along with a square matrix of permutation `P`, such that `L*U = P*A`.

➢ **Function**: `qr`

QR matrix factorization.

Syntax

```
(1)    [Q, R]    = qr(A)
```

Description

- `(1)` from the matrix `A`, returns an upper triangular square matrix in `R` and a unitary matrix in `Q` such that `A = Q*R`.

6.2.4 Matrix operations

➢ **Function**: `logm`

Matrix logarithm.

Syntax

```
(1)    L    = logm(A)
```

Description

- `(1)` return the logarithm of matrix `A`.
- by default, if the operand has real values, the type of the output is real. Use the `--complex` option to get complex values.

➢ **Function**: `kron`

Kronecker tension product.

Syntax

```
(1)    C = kron(A, B)
```

Description

- `(1)` returns the Kronecker tension product of matrices `A` and `B`.

6.2.5 Matrix structure

➢ **Function**: `tril`

Lower triangular part of matrix.

Syntax

```
(1)    C = tril(A)
(2)    C = tril(A,k)
```

Description

- `(1)` returns the lower triangular part of matrix `A`.
- `(2)` returns the element on and below the `k`-th diagonal of matrix `A`.

➢ **Function**: `triu`

Upper triangular part of matrix.

Syntax

```
(1)    C = triu(A)
(2)    C = triu(A,k)
```

Description

- (1) returns the upper triangular part of matrix A.
- (2) returns the element on and above the k-th diagonal of matrix A.

6.2.6 Matrix properties

➢ **Function**: cond

Condition number with respect to inversion.

Syntax

```
(1)    c = cond(A)
(2)    c = cond(A, 2)
(3)    c = cond(A, 1)
(4)    c = cond(A, inf)
(5)    c = cond(A, 'fro')
```

Description

- (1) and (2) return the 2-norm condition number of matrix A.
- (3) returns the 1-norm condition number of matrix A.
- (4) returns the infinity norm condition number of matrix A.
- (5) returns the Frobenius norm condition number of matrix A.
- in all versions, the condition number is a real.

➢ **Function**: condeig

Condition number with respect to eigenvalues.

Syntax

```
(1)    c          = condeig(A)
(2)    [V, D, c] = condeig(A)
```

Description

- (1) returns a vector of condition numbers for the eigenvalues of square matrix A, with numel(c) == size(A,1).
- (2) is equivalent to:

```
c = condeig(A)
[V, D] = eig(A)
```

➢ **Function**: det

Matrix determinant.

Syntax

```
(1)    d = det(A)
```

Description

- (1) returns the determinant of the square matrix A.

➢ **Function**: norm

Matrix norms.

Syntax

```
(1)    y = norm(V)
(2)    y = norm(V, p)
(3)    y = norm(V, inf)
(4)    y = norm(A)
(5)    y = norm(A, p)
(6)    y = norm(A, 'fro')
```

Description

- (1) returns the 2-norm or Euclidean norm of vector V
- (2) returns the vector norm defined by sum(abs(v)^p)^(1/p). p is a positive real.
- (3) returns max(abs(v)).
- (4) returns the 2-norm or maximum singular values of matrix A.
- (5) returns the p-norm of matrix A. p is 1, 2 or inf.
- (6) returns the Frobenius norm of the matrix A.
- in all versionq, the norm y is a real.

➢ **Function**: rank

Rank of matrix.

Syntax

```
(1)    r = rank(A)
(2)    r = rank(A, tol)
```

Description

- (1) returns the rank of the real matrix A. The rank is the number of singular values of A that are greater than the default tolerance max(size(A))*eps(norm(A)).
- (2) returns the rank of the real matrix A. The default tolerance is overrided by the real tol.

➢ **Function**: `rcond`

Reciprocal condition number.

Syntax

```
(1)    c = rcond(X)
```

Description

- `(1)` returns the reciprocal condition number in the 1-norm of the real square matrix `A`.

➢ **Function**: `trace`

Trace of matrix.

Syntax

```
(1)    y = trace(A)
```

Description

- `(1)` returns the trace of the square matrix `A`.

6.3 Random number generation

➢ **Function**: `rand`

Uniformly distributed random numbers.

Syntax

```
(1)    a = rand
(2)    A = rand(sz1, .., szN)
(3)    A = rand(n)
(4)    A = rand(V)
```

Description

- `(1)` returns a single real uniformly distributed random number in the interval (0,1).
- `(2)` create an `sz1`-by-...-by-`szN` array of random numbers. The `szi` parameters are integers.
- `(3)` create an `n`-by-`n` matrix of random numbers. `n` is an integer.
- `(4)` `V` is a vector of integers defining the size of `A`. This version is supported only if the number of elements of `V` is explicit.

Note

The rand function of eVariX™ uses the Mersenne Twister with a period of $2^{19937} - 1$ and an initial seed of 5489. This is the default configuration of some MATLAB® random generators, so unless the generator is changed, the results obtained with eVariX™ are the same as the results in MATLAB®.

➢ **Function**: `randn`

Normally distributed random numbers.

Syntax

```
(1)    a = randn
(2)    A = randn(sz1, .., szN)
(3)    A = randn(n)
(4)    A = randn(V)
```

Description

- `(1)` returns a single real normally distributed random number in the interval (0,1).
- `(2)` creates an `sz1`-by-...-by-`szN` array of random numbers. The `szi` parameters are integers.
- `(3)` creates an `n`-by-`n` matrix of random numbers. The `n` parameter is an integer.
- `(4)` `V` is a vector of integers defining the size of `A`. This version is supported only if the number of elements of `V` is explicit.

Note

The `randn` results are different between eVariX™ and MATLAB®.

➢ **Function**: `rng`

Random number generation control.

Syntax

```
(1)    rng(seed)
```

Description

- `(1)` sets the seed for the random number generator using the non-negative integer `seed` so that `rand`, and `randn` produce a predictable sequence of numbers.

➢ **Function**: `randperm`

Random permutation.

Syntax

```
(1)    V = randperm(n)
```

Description

- `(1)` returns a row vector containing a random permutation of the integers from 1 to `n` (inclusive).

Note

The `randperm` results are different between EVARIX™ and MATLAB®.

➢ **Function**: `randsample`

Random sample.

Syntax

```
(1)    V = randsample(n,k)
```

Description

- `(1)` returns a column vector containing `k` values sampled uniformly at random from the integers 1 to `n`.

Note

The `randsample` results are different between EVARIX™ and MATLAB®.

6.4 Interpolation

6.4.1 1-D interpolation

➢ **Function**: `interp1`

Linear interpolation.

Syntax

```
(1)    YP = interp1(X,Y,XP)
(2)    YP = interp1(X,Y,XP,method)
```

Description

- `(1)` returns the linear interpolation at specific query points.
- `(2)` also specifies the interpolation method.
  - `X` is a real vector containing the sample points;
  - `Y` is a real vector containing the corresponding values;
  - `XP` is a real array containing the coordinates of query points;
  - `YP` is a real array of the same size as `xp` containing the interpolated values.
  - `method` is a string. The only supported value is `'linear'`.

6.4.2 Gridded data interpolation

See also: ndgrid, meshgrid

➢ **Function**: `interp2`

Interpolation for 2-D gridded data in meshgrid format.

Syntax

```
(1)    Vq = interp2(V, Xq, Yq)
(2)    Vq = interp2(V, Xq, Yq, method)
```

Description

- `(1)` returns a real vector or matrix `Vq` of same size as `Xq` and `Yq`, containing interpolated values of a function of two variables at specific query points of interpolation.
  - `Xq` and `Yq` are vectors or arrays of real values defining the query points:
    - If `Xq` and `Yq` are real vectors of different orientations, then `Xq` and `Yq` are treated as grid vectors.
    - If `Xq` and `Yq` are real vectors of same orientation and size, then `Xq` and `Yq` are treated as scattered points in 2D space.
    - If `Xq` and `Yq` are real matrices (of same size), then they are treated as scattered points in 2D space.
  - `V` is a real matrix containing the sample grid points.
- `(2)` same as `(1)` with a string specifying the method of interpolation:
  - `'linear'` (default): linear interpolation;
  - `'cubic'`: cubic interpolation.

  ⚠ Limitation

  `'nearest'` and `'spline'` methods are not supported.
- This function can be used in `parallel block` annotations.

Gpu

- This function can be used in gpu sections and gpu `parallel blocks`.

6.5 Numerical Integration and Differentiation

➢ **Function**: `diff`

Differences and Approximate Derivatives.

Syntax

```
(1)    A = diff(B)
(2)    A = diff(B,n)
(3)    A = diff(B,n,dim)
```

Description

- (1) calculated the differences between the adjacent elements of matrix or 3D array of numeric values B along the first non-singleton dimension.
- (2) applies (1) n times.
- (2) applies (1) n times on the dim dimension of F.

n and dim must be integers. A and B are vectors, matrices of 3D-arrays of same size.

➢ **Function**: trapz

Trapezoidal numerical integration.

Syntax

```
(1)    Q = trapz(Y)
(2)    Q = trapz(Y,dim)
(3)    Q = trapz(X,Y)
(4)    Q = trapz(X,Y,dim)
```

Description

- (1) returns the approximate integral of array Y along the first non-singleton dimension.
- (2) returns the approximate integral of array Y along the dimension dim. dim is an integer.
- (3) returns the approximate integral of array Y with spacing increment specified by the vector X, along the first non-singleton dimension of Y.
- (4) same as (3) but works along the dimension dim.

6.6 Fourier Analysis and Filtering

6.6.1 Fourier transform

➢ **Function**: fft

Fast Fourier transform.

Syntax

```
(1)    F = fft(X)
(2)    F = fft(X,n)
(3)    F = fft(X,n,dim)
(4)    F = fft(X,[],dim)
```

Description

- (1) returns the discrete Fourier transform of X using a FFT algorithm.
  - if X is a vector, F is the Fourier transform of the vector.
  - if X is a matrix, fft considers the columns of X as vectors and returns the Fourier transform of each column.
  - if X is an array, fft considers the values along the first non-singleton dimension as vectors.
- (2) returns the n-point discrete transform of X. n is an integer.
  - if X is a vector, X is padded with 0 if n > numel(X), truncated to n elements otherwise.
  - if X is a matrix, each column is treated as in the vector case.

  - if X is an array, fft considers the values along the first non-singleton dimension as vectors, and the computation is done as in the vector case for each vector.
- (3) same as (2) but the Fourier transform in computed along the dimension dim. dim is an integer.
- (4) same as (1) but the Fourier transform in computed along the dimension dim. dim is an integer.
- Elements of X can be integer, real or complex values.
- Elements of F are always complex values.

➢ **Function**: fftshift

Shift zero-frequency component of center of spectrum.

Syntax

```
(1)    F = fftshift(X)
(2)    F = fftshift(X,dim)
```

Description

- (1) rearrange values of array X:
  - for vectors, swaps the left and right halves of X;
  - for matrices, swaps the first quadrant with the third and the second with the fourth;
  - for arrays, swaps half-spaces of X along each dimension.
- (2) applies the fftshift along the dimension dim. dim is an integer.

➢ **Function**: ifft

Inverse Fast Fourier transform.

Syntax

```
(1)    F = ifft(X)
(2)    F = ifft(X,n)
(3)    F = ifft(X,n,dim)
(4)    F = ifft(X,[],dim)
```

Description

- (1) returns the inverse discrete Fourier transform of X using an IFFT algorithm.
  - if X is a vector, F is the Fourier transform of the vector.
  - if X is a matrix, ifft considers the columns of X as vectors and returns the Fourier transform of each column.
  - if X is an array, ifft considers the values along the first non-singleton dimension as vectors.
- (2) returns the n-point inverse discrete transform of X. n is an integer.
  - if X is a vector, X is padded with 0 if n > numel(X), truncated to n elements otherwise.
  - if X is a matrix, each column is treated as in the vector case.
  - if X is an array, ifft considers the values along the first non-singleton dimension as vectors, and the computation is done as in the vector case for each vector.
- (3) same as (2) but the Fourier transform in computed along the dimension dim. dim is an integer.

- (4) same as (1) but the Fourier transform in computed along the dimension dim. dim is an integer.
- Elements of X can be integer, real or complex values.
- Elements of F are always complex values.

➢ **Function**: ifftshift

Inverse FFT shift

Syntax

```
(1)    F = ifftshift(X)
(2)    F = ifftshift(X,dim)
```

Description

- (1) undoes the result of fftshift.
- (2) applies the ifftshift along the dimension dim. dim is an integer.

6.6.2 Convolution

➢ **Function**: conv

Convolution.

Syntax

```
(1)    Z = conv(U,V)
```

Description

- (1) returns the convolution of U by V. U and V are scalars or vectors.

➢ **Function**: conv2

2D Convolution.

Syntax

```
(1)    C = conv2(A,B)
(2)    C = conv2(A,B,shape)
```

Description

- (1) returns the 2D convolution of matrix A by matrix B.
- (2) returns the 2D convolution of matrix A by matrix B with a string parameter shape defining how the convolution is done:
  - 'full' (default): computes the full 2D convolution.
    In this case, C is a matrix of size (size(A,1)+size(B,1)-1)-by-(size(A,2)+size(B,2)-1)
  - 'same': computes the central part of the convolution of same size as A;

  - `'valid'`: computes the convolution parts without the zero-padding of `A`. In this case, `C` is a matrix of size `max(size(A,1)-max(0,size(B,1)-1), 0)`-by-`max(size(A,2)-max(0,size(B,2)-1),0)`.
- This function can be used in `parallel block` annotations.

Gpu

- This function can be used in gpu sections and gpu `parallel blocks`.

6.6.3 Filtering

➢ **Function**: `filter2`

2D digital filter.

Syntax

```
(1)    Y = filter2(h,X)
(2)    C = filter2(h,X,shape)
```

Description

- `(1)` filters the data in the matrix `X` with the two-dimensional FIR filter in the matrix `h`.
- `(2)` same as `(1)` with a string parameter `shape` defining how the filtering is done:
  - `'full'` (default): computes the full 2D correlation.
  - `'same'`: computes the central part of the correlation.
  - `'valid'`: computes the correlations parts without the zero-padding of `X`.
- This function can be used in `parallel block` annotations.

Gpu

- This function can be used in gpu sections and gpu `parallel blocks`.

6.7 Statistics

6.7.1 Basic Statistics

➢ **Function**: `max`

Largest element in array.

Syntax

```
(1)    M = max(A)
(2)    M = max(A,[],dim)
(3) [M,I]= max(___)
(4)    M = max(A,x)
(5)    M = max(x,A)
(6)    M = max(A,B)
```

Description

- `(1)` returns the max of the elements of array `A` along the first non-singleton dimension.
- `(2)` returns the max of the elements of array `A` along the dimension `dim`. `dim` is an integer between 1 and `ndims(A)`.

- The size of `M` depends on the size of `A` and the dimension along which the max is done. This dimension is set to 1 in the resulting array.
- In some cases, eVarix™ does not have enough information to correctly compute the number of dimensions of`M`, and a warning is issued.
- `(3)` returns the max of the elements of `A` in array `M`, and their indices in the array of integers `I`. Parameters are the same as for `(1)` and `(2)`.
- `(4)` and `5` element-wise max between the scalar value `x` and the array `A`.
- `(6)` element-wise max between the element of arrays `A` and `B`. `A` and `B` have the same size.

Gpu

- This function can be used in gpu sections. Support for complex on gpu.

➢ **Function**: `mean`

Mean of array elements.

Syntax

```
(1)    B = mean(A)
(2)    B = mean(A,dim)
```

Description

- `(1)` returns the mean of the elements of array `A` along the first non-singleton dimension.
- `(2)` returns the mean of the elements of array `A` along the dimension `dim`. `dim` is an integer between 1 and `ndims(A)`.
- The size of `B` depends on the size of `A` and the dimension along which the mean is done. This dimension is set to 1 in the resulting array.
- In some cases, eVarix™ does not have enough information to correctly compute the number of dimensions of `B`, and a warning is issued.

Gpu

- This function can be used in gpu sections. Support for complex on gpu.

➢ **Function**: `median`

Median of array elements.

Syntax

```
(1)    B = median(A)
(2)    B = median(A,dim)
```

Description

- `(1)` returns the median of the elements of array `A` along the first non-singleton dimension.
- `(2)` returns the median of the elements of array `A` along the dimension `dim`. `dim` is an integer between 1 and `ndims(A)`.
- The size of `B` depends on the size of `A` and the dimension along which the median is done. This dimension is set to 1 in the resulting array.

- In some cases, eVariX™ does not have enough information to correctly compute the number of dimensions of `B`, and a warning is issued.

➢ **Function**: `min`

Smallest element in array.

Syntax

```
(1)    M = min(A)
(2)    M = min(A,[],dim)
(3) [M,I]= min(___)
(4)    M = min(A,x)
(5)    M = min(x,A)
(6)    M = min(A,B)
```

Description

- `(1)` returns the min of the elements of array `A` along the first nonsingleton dimension.
- `(2)` returns the min of the elements of array `A` along the dimension `dim`. `dim` is an integer between 1 and `ndims(A)`.
- The size of `M` depends on the size of `A` and the dimension along which the min is done. This dimension is set to 1 in the resulting array.
- eVariX™ does not have enough information to correctly compute the number of dimensions of `M`, and a warning is issued.
- `(3)` returns the min values in array `M` and their indices in the array of integers `I`. Parameters are same as in `(1)` and `(2)`.
- `(4)` and `5` element-wise min between the scalar value `x` and the scalar array `A`.
- `(6)` element-wise min between the elements of arrays `A` and `B`. `A` and `B` have the same size.

Gpu

- This function can be used in gpu sections.

➢ **Function**: `std`

Standard deviation.

Syntax

```
(1)    B = std(A)
(2)    B = std(A,w)
(3)    B = std(A,w,dim)
```

Description

- `(1)` returns the standard deviation of the elements of array `A` along the first non-singleton dimension.
- `(2)` returns the standard deviation of the elements of array `A` along the first non-singleton dimension. `w` is an integer controling how the result is normalized: by `N-1` if 0 (default), `N` if 1, where `N` is the number of values for computing the `std`.
- `(3)` returns the standard deviation of the elements of array `A` along the dimension `dim`. `dim` is an integer between 1 and `ndims(A)`.

- The size of `B` depends on the size of `A` and the dimension along which the computation is done. This dimension is set to 1 in the resulting array.
- In some cases, eVarix™ does not have enough information to correctly compute the number of dimensions of `B`, and a warning is issued.

➢ **Function**: `var`

Variance.

Syntax

```
(1)    B = var(A)
(2)    B = var(A,w)
(3)    B = var(A,w,dim)
```

Description

- `(1)` returns the variance of the elements of array `A` along the first non-singleton dimension.
- `(2)` returns the variance of the elements of array `A` along the first non-singleton dimension. `w` is an integer controling how the result is normalized: by `N-1` if 0 (default), `N` if 1, where `N` is the number of values for computing the `var`.
- `(3)` returns the variance of the elements of array `A` along the dimension `dim`. `dim` is an integer between 1 and `ndims(A)`.
- The size of `B` depends on the size of `A` and the dimension along which the computation is done. This dimension is set to 1 in the resulting array.
- In some cases, eVarix™ does not have enough information to correctly compute the number of dimensions of `B`, and a warning is issued.

6.7.2 Distributions

➢ **Function**: `normcdf`

Normal cumulative distribution function.

Syntax

```
(1)    n = normcdf(x)
(2)    N = normcdf(A)
```

Description

- `(1)` returns the normal cumulative distribution function at real value `x`.
- `(2)` returns the normal cumulative distribution function at elements of real array `A`. `B` is the same size as `A`.

➢ **Function**: `norminv`

Normal inverse cumulative distribution function.

Syntax

```
(1)    n = norminv(x)
(2)    N = norminv(A)
```

Description

- `(1)` returns the normal inverse cumulative distribution function at real value `x`.
- `(2)` returns the normal inverse cumulative distribution function at elements of real array `A`. `B` is the same size as `A`.

➢ **Function**: `normpdf`

Normal probability density function.

Syntax

```
(1)    n = normpdf(x)
(2)    N = normpdf(A)
```

Description

- `(1)` returns the normal probability density function at real value `x`.
- `(2)` returns the normal probability density function at elements of real array `A`. `B` is the same size as `A`.

CHAPTER 7
OTHER FUNCTIONS

7.1 IOs

➢ **Function**: `fprintf`

Print formatted strings.

Syntax

```
(1)    fprintf(1, format, ...)
(2)    fprintf(format, ...)
```

Description

- `(1)` and `(2)` displays a string formatted according to string `format` to the standard output.

⚠ Limitation

Printing to files is not yet supported.

➢ **Function**: `load`

Load variables from a file.

Syntax

```
(1)    load(filename)
```

Description

- `(1)` loads variable from the MAT-file filename.

During the generation of the executable eVariX™ doest not actually load the content of the file, but only the characteristics of the variables it defines, namely their scalar types and shapes. The values of the variables are not taken into account. If the content of `filename` is changed after the generation of the executable by eVariX™, it must define the same variables, with the same scalar types and shapes

as in the original file. In particular, the arrays must have the same number of dimensions, but can have different dimension sizes.

⚠ Limitation

- The loaded variables must be scalars or arrays. Structures and objects are not supported.
- Loading from an ascii file is not supported.
- If `filename` is not an explicit charater string, the `load` instruction must be preceded by a `load` annotation giving an explicit file name of a sample MAT-file (see page 54).

7.2 Signal Processing

➢ **Function**: `xcorr`

Cross-correlation.

Syntax

```
(1)    Z = xcorr(X,Y)
(2)    Z = xcorr(X,Y,lag)
```

Description

- `(1)` returns the cross-correlation of two discrete-time sequences stored in real vectors `X` and `Y`.
- `(2)` limits the lag range from `-lag` to `lag`.
  - `Z` is a vector of size `2*max(numel(X),numel(Y))-1`

➢ **Function**: `convmtx`

Convolution matrix.

Syntax

```
(1)    M = convmtx(V,n)
```

Description

- `(1)` returns the matrix `M` such that the product of `M` and the vector `X`, is the convolution of `V` and `X`.
  - `n` is a numerical value.
  - if `V` is a column vector of length m, the size of `M` is $(m+n-1)$-by-n.
  - if `V` is a row vector of length m, the size of `M` is n-by-$(m+n-1)$.

➢ **Function**: `hamming`

Hamming window.

Syntax

```
(1)    w = hamming(L)
```

Description

- (1) returns an L-point symmetric Hamming window.
  - L is a positive integer.

➢ **Function**: levinson

Levinson-Durbin recursion.

Syntax

```
(1)         a  = levinson(r)
(2)         a  = levinson(r,n)
(3)       [a,e] = levinson(r,n)
(4)     [a,e,k] = levinson(r,n)
```

Description

- (1) returns the coefficients of a $length(r) - 1$ order auto-regressive linear process which has r as its auto-correlation sequence.
- (2) returns the coefficients for an auto-regressive model of order n.
- (3) returns the prediction error e of order n.
- (4) returns the reflection coefficients k as a column vector of length n.
  - r is a real or complex deterministic auto-correlation sequence.
  - if r is a matrix, levinson finds the coefficients for each column of r and returns them in the rows of a.
  - n is a real.

7.3 Image Processing

➢ **Function**: imnoise

Add noise to image.

Syntax

```
(1)     J = imnoise(I,type)
```

Description

- (1) add noise of a given type to the image I.
  - I is a matrix of int8.
  - type is a string equal to 'gaussian'.

⚠ Limitation

Only Gaussian noise is supported.

Note

Computation of noise in based on randomly chosen values. The results of the eVariX™ function may differ from the results of MATLAB®.

➢ **Function**: `imshow`

Display image.

Syntax

```
(1)    imshow(I)
```

Description

- `(1)` displays the image `I`. `I` is a matrix of `int8` for grayscale images or a 3D-array of reals with the third dimension equal to 3 for RGB images, each value in this dimension representing one RGB color.

⚠ Limitation

- This function requires the use of the OpenCV library (see corresponding section in the documentation of the compilation process).
- This function is only available on Linux systems.

➢ **Function**: `imread`

Read image from graphics file.

Syntax

```
(1)    I = imread(filename)
(2)    I = imread(filename, i)
```

Description

- `(1)` reads the image from the file specfied by the string `filename`. `I` is a matrix of `int8` values and has the size of the image.
- `(2)` reads the `i`-th image from the TIF file specfied by the string `filename`. `I` is a matrix of `int8` values and has the size of the image.

⚠ Limitation

- Only grayscale TIF images are supported.
- This function requires the use of the OpenCV library (see corresponding section in the documentation of the compilation process). The second variant requires at least OpenCV 3.3.
- This function is only available on Linux systems.

➢ **Function**: `padarray`

Pad array

Syntax

```
(1)    B = padarray(A, p)
(2)    B = padarray(A, V)
(3)    B = padarray(A, p, method)
(4)    B = padarray(A, V, method)
```

Description

- (1) returns the numeric matrix `A` padded with 0 on both sides of the first dimension such as `size(B,1) = size(A,1) + 2*p`. `p` is an integer.
- (2) returns the numeric matrix `A` padded with 0. The integer vector `V` specifies the size of the padding on each dimension, such as `size(B,i) = size(A,i) + 2*V(i)`.
- (3) and (4) same as above, but with a string `method` specifying how the padding is performed.

⚠ Limitation

Only the `'symmetric'` method is supported.

7.4 Performance

➢ **Function**: `tic`

Start clock to measure performance.

Syntax

```
(1)    tic
(2)    timer = tic()
```

Description

- (1) starts the CRT internal stopwatch timer.
- (2) returns a real representing the time at execution of the `tic`.

➢ **Function**: `toc`

Read elapsed time from stopwatch.

Syntax

```
(1)    toc
(2)    t = toc(timer)
```

Description

- (1) read the CRT internal stopwatch timer and displays the elapsed time.
- (2) returns the elapsed time since the `tic` command corresponding to the real `timer`.

Part III

Appendix

CHAPTER 8

EVARIX™- END USER LICENSE AGREEMENT

The eVariX™ product and its modules, namely the Cold© compiler, the `evarix` driver and the CRT library, are the property of SILKAN (see section 4). By using eVariX™ Softwares, the user accepts the following terms and conditions.

1 LIMITS

Licensee agrees not to (directly or indirectly, and in whole or in part):

(a) make copies of the SILKAN Products,

(b) provide access to the SILKAN Products to anyone other than Licensee's employees, contractors, or consultants under written contract with Licensee agreeing to be bound by terms at least as protective of SILKAN as those in this license;

(c) use the license on another Hardware than the Specific Hardware on which the license has been implemented,

(d) sublicense, distribute, pledge, lease, rent, or commercially share (including timeshare) the SILKAN Products or any of Licensee's rights herein;

(e) use the SILKAN Products for purposes of providing a service bureau, including, without limitation, providing third-party hosting, or third-party application Integration or application service provider-type services, or for any similar services;

(f) use the SILKAN Products in connection with any ultra-hazardous activity, or any other activity for which its failure might result in serious property damage, or death or serious bodily injury; or

(g) modify, translate, reverse engineer, decrypt, decompile, disassemble, create derivative works based on, or otherwise attempt to discover the SILKAN Products source code or underlying ideas or algorithms.

2 LIMITED WARRANTIES

THE SILKAN PRODUCTS AND SERVICES ARE PROVIDED "AS IS', AND ALL OTHER EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS, AND WARRANTIES INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (EVEN IF INFORMED OF SUCH PURPOSE), OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE, ARE HEREBY EXCLUDED TO THE EXTENT ALLOWED BY APPLICABLE LAW. CERTAIN THIRD PARTY SOFTWARE MAY BE PROVIDED TO LICENSEE ALONG WITH CERTAIN SILKAN PRODUCTS AS AN ACCOMMODATION TO LICENSEE. THIS THIRD PARTY SOFTWARE IS PROVIDED "AS IS'. LICENSEE MAY CHOOSE NOT TO USE THIRD

PARTY SOFTWARE PROVIDED AS AN ACCOMMODATION BY SILKAN. NO WARRANTY IS MADE THAT THE SILKAN PRODUCTS'S FUNCTIONALITY OR SERVICES WILL MEET LICENSEE'S REQUIREMENTS, OR THAT THE OPERATION OF THE SILKAN PRODUCTS OR SERVICES WILL BE UNINTERRUPTED OR ERROR-FREE.

3 LIMITATION OF LIABILITY

IN NO EVENT WILL SILKAN BE LIABLE FOR ANY LOST DATA, LOST REVENUE, LOST PROFITS, DAMAGE TO REPUTATION, BUSINESS INTERRUPTION, OR ANY INDIRECT, INCIDENTAL, CONSEQUENTIAL, SPECIAL, PUNITIVE, EXEMPLARY OR ANY SIMILAR TYPE OF DAMAGES ARISING OUT OF THE USE OR THE INABILITY TO USE THE SOFTWARE, OR THE PROVISION OF ANY SERVICES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

4 TITLE AND PROPRIETARY NOTICES

SILKAN Products are proprietary to SILKAN and are protected by applicable French and international patent, copyright, trademark and trade secret laws ("Intellectual Property'). SILKAN and its licensors shall retain title to the SILKAN Products and all intellectual Property (and any other rights) embodied therein. All proprietary notices incorporated in or affixed to any SILKAN Products or Documentation shall be duplicated by Licensee on all copies of the SILKAN Products or Documentation, as applicable, and shall not be altered, removed or obliterated.

5 JURISDICTION AND VENUE

All disputes arising out of or related to this Agreement, whether based on contract, tort, of any legal or equitable theory, will be subject to the exclusive jurisdiction of the court of the Tribunal de Commerce de Paris.

6 LICENSES

The scope and term of license depends on the type of license the licensee are provided by SILKAN.

6.1 Free License

A free license is granted by SILKAN for a strictly limited period (LIMITED LICENSE) and a specific computer (NODE-LOCKED LICENSE). Using a free license, licensee may use the product for internal evaluation purposes and only for the term of the validity period. This kind of license permits ONLY a noncommercial use of the materials generated by EVARIX programs. In particular the redistribution of the CRT library is strictly forbidden.

6.2 Commercial License

Commercial licenses are subject to specific agreements and/or contracts between SILKAN and the licensee.
Contact evarix.product@silkan.net for more details.

CHAPTER 9

THIRD PARTIES TOOLS - LICENCES AND DISCLAIMERS

1 The ISL library

Depending on your version of eVariX™, it may be linked to the ISL library, which is provided under the MIT licence:

```
MIT License (MIT)

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to do
so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

2 The Eigen library

Depending on your version of eVariX™, it may be distributed with the Eigen Library, which is released under the MPL 2.0 licence, available from:

https://www.mozilla.org/en-US/MPL/2.0/

Notice in particular the *Disclaimer or Warranty*:

```
Covered Software is provided under this License on an "as is" basis,
without warranty of any kind, either expressed, implied, or statutory,
including, without limitation, warranties that the Covered Software is
free of defects, merchantable, fit for a particular purpose or
non-infringing.  The entire risk as to the quality and performance of
```

```
the Covered Software is with You. Should any Covered Software prove
defective in any respect, You (not any Contributor) assume the cost of
any necessary servicing, repair, or correction. This disclaimer of
warranty constitutes an essential part of this License. No use of any
Covered Software is authorized under this License except under this
disclaimer.
```

and the *Limitation of Liability*:

```
Under no circumstances and under no legal theory, whether tort
(including negligence), contract, or otherwise, shall any Contributor,
or anyone who distributes Covered Software as permitted above, be
liable to You for any direct, indirect, special, incidental, or
consequential damages of any character including, without limitation,
damages for lost profits, loss of goodwill, work stoppage, computer
failure or malfunction, or any and all other commercial damages or
losses, even if such party shall have been informed of the possibility
of such damages. This limitation of liability shall not apply to
liability for death or personal injury resulting from such party's
negligence to the extent applicable law prohibits such
limitation. Some jurisdictions do not allow the exclusion or
limitation of incidental or consequential damages, so this exclusion
and limitation may not apply to You.
```

The source code is available from:

http://eigen.tuxfamily.org

3 The Boost.Compute library

Depending on your version of eVariX™, it may be distributed with the a derivative of the Boost.Compute library, which is released under the following Boost Software License:

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

4 The clBlas library

Depending on your version of eVariX™, it may be distributed with the the clBlas library, which is released under the following Apache License:

the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained

within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

```
   9. Accepting Warranty or Additional Liability. While redistributing
      the Work or Derivative Works thereof, You may choose to offer,
      and charge a fee for, acceptance of support, warranty, indemnity,
      or other liability obligations and/or rights consistent with this
      License. However, in accepting such obligations, You may act only
      on Your own behalf and on Your sole responsibility, not on behalf
      of any other Contributor, and only if You agree to indemnify,
      defend, and hold each Contributor harmless for any liability
      incurred by, or claims asserted against, such Contributor by reason
      of your accepting any such warranty or additional liability.

   END OF TERMS AND CONDITIONS
```

5 The PAPI library

Depending on your version of eVariX™, it may be distributed with the the PAPI library[1], which is released under the following license:

```
Copyright © 2017 The University of Tennessee. All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:
· Redistributions of source code must retain the above copyright notice,
 this list of conditions and the following disclaimer.
· Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer listed in this license
  in the documentation and/or other materials provided with the distribution.
· Neither the name of the copyright holders nor the names of its contributors
  may be used to endorse or promote products derived from this software without
  specific prior written permission.

This software is provided by the copyright holders and contributors "as is"
and any express or implied warranties, including, but not limited to, the implied
warranties of merchantability and fitness for a particular purpose are disclaimed.
in no event shall the copyright owner or contributors be liable for any direct,
indirect, incidental, special, exemplary, or consequential damages (including,
but not limited to, procurement of substitute goods or services; loss of use,
data, or profits; or business interruption) however caused and on any theory
of liability, whether in contract, strict liability, or tort (including negligence
or otherwise) arising in any way out of the use of this software, even if advised
of the possibility of such damage.
```

6 The Elfio library

Depending on your version of eVariX™, it may be distributed with the the Elfio library, which is released under the following MIT license:

```
MIT License

Copyright (C) 2001-2011 by Serge Lamikhov-Center

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
```

[1] http://icl.cs.utk.edu/papi/software/index.html

```
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

INDEX