Snippet user manual

Version 1.02

Riccardo Scorretti
riccardo.scorretti@ec-lyon.fr

Index

I. Introduction

I.A. What is Snippet?

Snippet is a Matlab toolbox, which allows to “plug” short portions of code in C/C++ in Matlab programs, for instance:

```
x = 1:10;

% This is a C snippet
snippet({ ...
  '#pragma input x(N)', ...
  'for(size_t i=0 ; i<N ; ++i) x[i] = -x[i]'}, 'langage', 'c');

disp(x);
```

I.B. How does it work?

Starting from small C/C++ source codes (= snippets), Snippet generates a `.m` gateway function, and a complete `.c` or `.cpp` source file which includes all the necessary code to retrieve input arguments from Matlab, and to provide output arguments, so that developers can focus only on the computational part of their algorithms, leaving to Snippet the boring and error-prone task to transfer input/output arguments from/to Matlab.
The C/C++ snippets can be stored either in an external `.c` or `.cpp` text file, or in a cell-array. The generated source and compiled code are written in a directory which is user-specific.

I.C. Requirements

Snippet requires a working Mex compilation environment, configured to be able to compile C/C++ source codes. Snippets are compiled only when necessary, that is if their source code has been modified, or simply if they never have been compiled. However, it is possible to force Snippet to recompile the codes in several ways:

- by using the static method `snippet.clear`: this method deletes the compiled codes for all snippets;
- by using the method `deleteBinary`: this method deletes the compiled code for a particular snippet;
- by passing the optional arguments `'compile', true` to the constructor.

I.D. Limitations

At present time, only numerical data (scalars, vectors or matrix or any dimension) are handled. In particular:

- strings, objects and cell-array are not handled,
- sparse matrix are not handled as well,
- support for complex arithmetic in C++ is still not working,
- snippets in C must be compiled with the option `-R2018a`, and will not work with previous versions.

I.E. Caveat

Please notice that at present time Snippet is still in a beta version, and hence it probably still contains many bug or undesired behaviour. Use it at your own risk.

I.F. License

Snippet is distributed under licence GNU Lesser General Public License v3.0 or later:
https://www.gnu.org/licenses/lgpl-3.0.en.html

II. Quick start

II.A. Hello, world

Assume that you have a vector (namely `x`) and you want to write a C snippet which compute the sum of the terms of `x`, multiplies each term by a factor 2 and stores the result in the new variables named respectively `acc` and `out`:

```
x = (1:5);
snippet[1]({ ...
  '#pragma input x(N)[2]', ...
  '#pragma output out:double(N), acc:double[3]', ...
  'acc=0.0;', ...
  'for(size_t i=0 ; i<N ; ++i) {', ...
  '  acc += x[i][4];', ...
  '  out[i] = 2.0*x[i];', ...
  '}', ...
  }, 'language', 'c'[5]);
disp(out[6])
disp(acc)
```

This snippet provides the following result:

1 This instruction creates an anonymous instance of `snippet`, compile it (if necessary) and executes it.

2 Input arguments are provided by the directive `#pragma input`. The value of input arguments is retrieved automatically from caller's workspace and passed to the compiled Mex function. In this case, the variable `x`, which is expected to be a vector of size `N` is passed to the snippet. The type of `x` is inferred automatically, but it is better (= safer) to declare it explicitly (see below).

3 Output arguments are provided by the directive `#pragma output`. The synopsis is identical to the one of the directive `#pragma input`. The type of output arguments cannot be inferred automatically because variables `acc` and `out` don't exist yet, or may be used to store data of different types. Hence, it is strongly advised to declare explicitly their type, which in this case is `double`. The default type is `double`. When more than one arguments have to be provided, arguments must be separated by a coma(`,`).

4 There is no need to declare input and output arguments in a C/C++ style: Snippet will do the job. Their value is automatically retrieved and passed to the compiled Mex function.

5 The default programming language is C++. If you want to write C snippets, it is mandatory to specify the option `'language', 'c'`.

6 Similarly, the value of output arguments `acc` and `out` is automatically transferred to the caller's workspace by Snippet.

```
>> x = (1:5);
>> snippet({ ...
  '#pragma input x(N)', ...
  '#pragma output out:double(N), acc:double', ...
  'acc=0.0;', ...
  'for(size_t i=0 ; i<N ; ++i) {', ...
  '  acc += x[i];', ...
  '  out[i] = 2.0*x[i];', ...
  '}', ...
  }, 'language', 'c');
>> disp(out)
     2
     4
     6
     8
    10

>> disp(acc)
    15
```

Notice that the variable `out` is a column vector, whereas the input argument `x` is a row vector (in fact, it is casted to a column vector before the compiled Mex function is invoked). The variables `out` and `acc` are automatically created when the snippet is executed.

II.B. Snippets are compiled only when required

The very first time the snipped is executed it has to be compiled, and this requires a certain time. Any following execution of the snippet will invoked the compiled version (even if you restart Matlab) unless the snippet has been modified, or if binaries have been deleted, for instance by invoking the method `snippet.clear`.

Compiled snippets are stored in a directory which depends on the user. It is possible to know where snippets are stored by invoking the method `snippet.info`:

```
>> snippet.info
Snippet version                : 1.0
Path where mex-files are stored: /home/scorretti/.snippets
---
Snippet is distributed under the licence LGPL-3.0-or-later

Checking the compiler:
 - MEX functions in C   can be compiled
 - MEX functions in C++ can be compiled
```

Notice that the method `info` also checks that C/C++ Mex compilers are correctly installed and configured.
If one has a look to the directory where compiled snippets are stored, one would see many files with strange names, for instance:

```
>> dir /home/scorretti/.snippets


.
```

```
..
sngtw_29b27021c29a28d24f1f81a91a01d022165x.m
sngtw_2f5c.m
sngtw_2f5x.m
sngtw_40238642643536c36041d29f2a72ff348303c.m
sngtw_4774363cd4d03ba36128c3d12b23833ff19fx.m
sngtw_5f95f752959a61d4ba5233ff5624143f0485x.m
snmex_29b27021c29a28d24f1f81a91a01d022165x.cpp
snmex_29b27021c29a28d24f1f81a91a01d022165x.mexa64
snmex_2f5c.c
snmex_2f5c.mexa64
snmex_2f5x.cpp
snmex_2f5x.mexa64
snmex_40238642643536c36041d29f2a72ff348303c.c
snmex_40238642643536c36041d29f2a72ff348303c.mexa64
snmex_4774363cd4d03ba36128c3d12b23833ff19fx.cpp
snmex_4774363cd4d03ba36128c3d12b23833ff19fx.mexa64
snmex_5f95f752959a61d4ba5233ff5624143f0485x.cpp
snmex_5f95f752959a61d4ba5233ff5624143f0485x.mexa64
```

This is because each snippet is associated with a particular name, which is generated basing on the source code. If the source code is modified, this name is very likely to change. This is the way Snippet checks that a particular snippet has been modified. Of course, this is not 100% save, because collisions[7] may eventually happen. However, notice that that snippets have to be compiled only once, and it is possible to force the compilation of snippets by providing the optional argument `'compile', true`.
Also, it has to be observed that this control requires some times. However, if you make the effort to write a snippet in C/C++ it is to spare *a lot* of computational time, and hence in practice the time "wasted" to check if a snippet has to be recompiled is insignificant.

II.C. Anonymous snippets

In the previous example the snippet was anonymous, that is it is not assigned to any variable. In fact, snippets are objects and hence can be assigned to a variable and passed to functions like any other object. For instance, one could write:

```
>> x = (1:5);
>> mySnippet = snippet({ ...
  '#pragma input x(N)', ...
  '#pragma output out:double(N), acc:double', ...
  'acc=0.0;', ...
  'for(size_t i=0 ; i<N ; ++i) {', ...
  '  acc += x[i];', ...
  '  out[i] = 2.0*x[i];', ...
  '}', ...
  }, 'language', 'c');
>> mySnippet.run();[8]
```

In this case, the snippet is stored in the variable `mySnippet`, and hence it is not anonymous. The execution of snippets can be triggered at any time by using the method **`run`**.

7 That is, different snippets which are associated with exactly the same name.
8 The method `run` compiles the snippet (if required) and runs it.

II.D. Editing the generated code manually

Snippets which are stored in variables provide the possibility to edit and modify their code more easily than anonymous snippets. In order to display the code in the console, the method **list** can be used:

```
>> mySnippet.list

#pragma input x(N)
#pragma output out:double(N), acc:double
acc=0.0;
for(size_t i=0 ; i<N ; ++i) {
  acc += x[i];
  out[i] = 2.0*x[i];
}
```

This displays the code of the snippet, which in fact is not very useful. However, it is possible to edit the complete **.m** and **.c** (or **.cpp**) source code by using the method **edit**:

```
>> mySnippet.edit
```

This method opens the files associated with the snippet in Matlab editor. Two files are generated for each snippet: a **.m** gateway function, and the associated C/C++ source Mex file.

The gateway function corresponding to our first example is listed hereafter. The purpose of the gateway is to retrieve the values of input arguments, convert them to the declared type, run the compiled Mex function and create the variables corresponding to output arguments into the caller's workspace.

```
function sngtw_40238642643536c36041d29f2a72ff348303c[9]()
% Generated automatically by snippet v 1.0

% Retrieve the input arguments from the caller workspace
x = evalin('caller', 'x');[10]
x = x(:);[11]
x = double(x);

% Call the compiled MEX function
[out, acc] = snmex_40238642643536c36041d29f2a72ff348303c(x);

% Assign the output arguments in the caller workspace
assignin('caller', 'out', out);[12]
assignin('caller', 'acc', acc);
end
```

The complete C source function of the snippet is listed hereafter. One observes that the "naked" code of the snippet (apart from **#pragmas**) is found in lines 16-20: all the rest of the source code, which is required to transfer input/output arguments from/to Matlab is generated automatically.

9 The strange names of these files depend on the source code of the snippet, and are used to check if snippets need to be compiled.
10 The value of input arguments are retrieved from the caller's workspace.
11 Row vectors are systematically casted to column vectors. Moreover, each input argument is casted to the specified type before they it is passed to the compiled Mex function.
12 The variables corresponding to output arguments are created in the caller's workspace.

```
01: // C MEX function generated automatically by snippet v 1.0
02:
03: #include "mex.h"
04: #include "matrix.h"
05:
06:
07: void mexFunction( int nlhs, mxArray *plhs[],
08:                   int nrhs, const mxArray *prhs[])
09: {
10:     size_t N;
11:   N = mxGetM(prhs[0]);
12:   mxDouble *x = mxGetDoubles(prhs[0]);
13:   plhs[0] = mxCreateNumericMatrix(N, 1, mxDOUBLE_CLASS, mxREAL);
14:   mxDouble *out = (mxDouble *) mxGetDoubles(plhs[0]);
15:   mxDouble acc;
16:     acc=0.0;[13]
17:     for(size_t i=0 ; i<N ; ++i) {
18:       acc += x[i];
19:       out[i] = 2.0*x[i];
20:     }
21:   plhs[1] = mxCreateDoubleScalar(acc);
22: }
```

In order to debug snippets, or simply to have a better understanding of Matlab API, it may be desirable to manually modify the source code. The modified code can be compiled by using the method **`compileMex`**. For instance, one may want to add a command to display a message before line 16:

```
16:     mexPrintf("*** Here the snippet is executed ***\n");
```

Then, to recompile the edited (= modified) version of the C/C++ code and run it, one uses the following commands:

```
>> mySnippet.compileMex
>> >> mySnippet.run
*** Here the snippet is executed ***
```

Warning: if you force recompilation the snippet, for instance by using the method **`compile`**, all modifications manually edited are irreversibly lost, because source files are regenerated by Snippet.

II.E. Snippet can be used to generate functions

In the previous examples, the source code of snippets was "inlined" in Matlab code. Another possible usage of Snippet is to generate C/C++ Mex functions which can be distributed independently from their source code. To this aim, it is enough to create a (non anonymous) snippet, and to invoke the method **`compile`** by providing the name of the function which has to be generated.

The following example illustrates how to create a function named **`binsearch`** which implements the binary search in a sorted array of **`double`**. In this case, the "naked" code of the snippet has been created by using a

13 Here it can be observed the "naked" source code of the snippet.

text editor and stored in the file **binsearch.c**. Notice that this is not mandatory at all, but in the case of not-so-short snippets it is much more comfortable to use a text editor to write the code.

```c
// binsearch.c:

#include <stdint.h>

#pragma input x:double(N), key:double
#pragma output ind:double, tf:uint8

uint64_t lb=0, ub=N-1, p;
tf = 0;
while(lb <= ub) {
  p = (lb+ub)/2;
  if(x[p] < key) {
    lb = p + 1;
  } else if (x[p] > key) {
    ub = p - 1;
  } else {
    tf = 1;
    break;
  }
}
ind = (double) (p+1);
```

The procedure to create the Mex function is the following:

```
>> mySnippet = snippet('binsearch.c');
>> mySnippet.compile('binsearch');
```

Notice that in this case it is no more necessary to specify the programming language: Snippet infers it basing on the extension `.c` of the source file. On my PC, after the compilation, the following files are generated:

```
>> dir binsearch*

binsearch.c        binsearch_.c
binsearch.m        binsearch_.mexa64
```

The function **binsearch.m** is the gateway function, **binsearch_.c** is the corresponding C source code and **binsearch_.mexa64** is the executable[14]. The gateway function and the executable can be distributed, without sharing the complete source code which is stored in **binsearch_.c**. The function **binsearch** is ready to be executed:

```
>> vec = 1:10;
>> [ind, flag] = binsearch(vec, 6)
ind =
     6

flag =
  uint8

   1
```

14 Needless to say, the extension of executables is system dependent.

III. Specificities of C snippets

III.A. A modern (= C99) compiler is required

To begin with, Snippet generates C source codes which requires a "modern" C99 compiler. In particular, Variable-Length-Array (VLA) are used to allow a simpler way to handle matrix. Nowadays all major compilers implement C99 standard, hence this should not be an issue.

III.B. Input arguments may be modified directly

Apart from the case of scalar input arguments, Matlab C API provides a direct access to the memory where data is stored. A practical consequence of this fact is that *in principle* modifications of input arguments are permanent, apart from the case of scalar input[15]. That is, when a variable is passed as input argument to a snippet written in C, it can be modified directly. For instance:

```
>> x = 1 : 5;
>> snippet({...
     '#pragma input x(N)', ...
     'for(int i=0 ; i<N ; ++i) ++x[i];' ...
   }, 'language', 'c');
>> x

x =


     2     3     4     5     6
```

Warning: this practice is undocumented, and hence discouraged, and could lead to wrong and/or unpredictable results. However, it is important to be aware of such a possible side effect.
Unless it is absolutely necessary to maximize the speed (by avoiding to make a copy of data), it is more prudent to avoid to rely on direct modification of input arguments.

In fact this point is particularly tricky. Normally, input arguments can indeed be modified from within C Mex functions. The key is that in the gateway function, which is automatically generated by Snippet, some (apparently) harmless manipulations which modify the address where values are stored *may, or may not be* executed. For instance:

```
>> format debug
>> z = (1:3) + i

z =


Structure address = 7fbd3ceff400
m = 1
```

15 This exception is due to the particular way Snippet handles scalar input arguments.

```
n = 3
pr = 7fbd076847c0

   1.0000 + 1.0000i   2.0000 + 1.0000i   3.0000 + 1.0000i

>> z = z(:)

z =


Structure address = 7fbd3c390940
m = 3
n = 1
pr = 7fbbef6c3000

   1.0000 + 1.0000i
   2.0000 + 1.0000i
   3.0000 + 1.0000i
```

Notice that the address where the values of **z** are stored changed before and after the instruction **z = z(:)**. This seems not to be the case for real-valued variables.

III.C. On the way matrix are handled

In order to simplify the way matrix are handled, Variable Length Array are used by Snippet. However, there is an issue due to the fact that Matlab stores matrix column-wisely (= like in FORTRAN) whereas C assumes a row-wise storage. In order to illustrate this concept let's examine the following example which takes a 2D matrix **x** as input and returns the transposed matrix **tx**:

```
S = snippet({ ...
  '#pragma input x:double(M,N)[16]', ...
  '#pragma output tx:double(N,M)', ...
  '', ...
  '#define X(i,j) x[j][i][17]', ...    % *** Pay attention to the order of index i and j ***
  '#define TX(i,j) tx[j][i]', ...
  '', ...
  'for(int i=0 ; i<N ; ++i) {', ...
  '  for(int j=0 ; j<M ; ++j) {', ...
  '    TX(i,j) = X(j,i)[18];', ...
  '  }', ...
  '}', ...
  }, 'language', 'c');
```

For instance:

```
>> x = magic(3)

x =
```

16 **x** is a 2D matrix of **double**, the size of which is stored in variables **M**, **N**.

17 In order to simplify handling of matrix, it may be practical to define a parametric macro for each matrix, where the order of index is inverted. This holds also for N-dimension matrix.

18 By using macros, matrix can be handled seamlessly.

```
     8     1     6
     3     5     7
     4     9     2

>> S.run
>> tx

tx =

     8     3     4
     1     5     9
     6     7     2
```

The key is in the way the variables `x` and `tx` are declared and initialized. For instance, in the case of `x` we find:

```
13:   M = mxGetM(prhs[0]);
14:   N = mxGetN(prhs[0]);
15:   mxDouble (*x)[M] = (mxDouble (*)[M]) mxGetDoubles(prhs[0]);
```

The most interesting line is 15, where `x` is defined as a pointer over an array of vectors of size `M`. Notice that the size `M` is known at run-time: this is possible thanks to the nice C99 VLA feature[19]. This allow to access to the terms of the matrix `x` in the following way:

`x[j][i]`

In particular, this mechanism avoids to the user to compute "by hand" the address (or the index) of each term, like in C89.

As anticipated, the only complication is that Matlab stores matrix column-wise, where C uses row-wise storage. Hence, in practice the order of index has to be inverted! That is, one has to write `x[j][i]` whereas the "natural" (= intuitive) way would have been to write `x[i][j]`. In order to avoid such a weird programming style, it may be very practical to define a parametric macro for each matrix, so as to program computations in a more intuitive way:

`X(i,j)`

Notice that this issue is not due to the C language. For instance, consider the following C program, where the matrix is accessed by using index in the natural (= intuitive) order:

```
#include <stdlib.h>
#include <stdio.h>

void print(size_t M, size_t N, float *buffer)
{
  float (*mat)[N] = (void *) buffer;

  for(size_t i=0 ; i<M ; ++i) {
    for(size_t j=0 ; j<N ; ++j) {
      printf("%5.1f  ", mat[i][j]);
    }
    putchar('\n');
  }
```

19 From my personal standpoint, it is regrettable that VLA have not been included in C++.

```
}

int main()
{
  float mat[4][5] = {
    { 1.0,  2.0,  3.0,  4.0,  5.0},
    { 6.0,  7.0,  8.0,  9.0, 10.0},
    {11.0, 12.0, 13.0, 14.0, 15.0},
    {16.0, 17.0, 18.0, 19.0, 20.0} };

  print(4, 5, (float *) mat);
  return 0;
}
```

```
$ gcc mat.c -Wall -W -std=c99 -o mat
$ ./mat
  1.0    2.0    3.0    4.0    5.0
  6.0    7.0    8.0    9.0   10.0
 11.0   12.0   13.0   14.0   15.0
 16.0   17.0   18.0   19.0   20.0
```

III.D. On the usage of complex arithmetic

Snippet supports only interleaved complex API, that is the API implemented starting from Matlab 2018a. With this API each complex number is stored in a structure composed of the fields **`.real`** and **`.imag`**, which store respectively the real and imaginary part. As an example, consider the following program which computes the conjugate of a complex vector **`z`**:

```
z = rand(4,1) + i*rand(4,1);
disp(z);
snippet({ ...
  '#pragma input z:complex(N)', ...
  '#pragma output cz:complex(N)', ...
  'for(size_t i=0 ; i<N ; ++i) {', ...
  '  cz[i].real =  z[i].real;', ...
  '  cz[i].imag = -z[i].imag;', ...
  '}' ...
  }, 'language', 'c');
disp(cz);
```

```
z =
   0.4035 + 0.7051i
   0.9350 + 0.5586i
   0.4795 + 0.7566i
   0.2318 + 0.9955i

cz =
   0.4035 - 0.7051i
   0.9350 - 0.5586i
   0.4795 - 0.7566i
   0.2318 - 0.9955i
```

IV. Specificities of C++ snippets

IV.A. Input arguments cannot be modified directly

Conversely to the case of C snippets, it is not easy (or at least it is more tricky) to modify input arguments from a decently written C++ Mex functions. This is due to the fact that Matlab implements a copy-on-write mechanism. However, conversely to the case of C snippets[20], it is allowed to define the very same variable both as input and output argument. For instance:

```
x = 1:3
snippet({ ...
  '#pragma input x(N)', ...
  '#pragma output x(N)', ...
  '', ...
  'for(int i=0 ; i<N ; ++i) x[i] = 2.0*x[i];' ...
  });
disp(x)
```

```
       2
       4
       6
```

IV.B. On the way matrix are handled

Conversely to C snippets, the difference between Matlab and C in the way matrix are stored is handled directly by Matlab C++ API. Hence, manipulating matrix in C++ snippets is much less tricky; in particular, the order of index is not an issue. For instance, the C++ version of the snippet which transposes a 2D matrix writes (cf. III.C):

```
snippet({ ...
  '#pragma input x:double(M,N)', ...
  '#pragma output tx:double(N,M)', ...
  '', ...
  'for(int i=0 ; i<N ; ++i) {', ...
  '  for(int j=0 ; j<M ; ++j) {', ...
  '    tx[i][j] = x[j][i][21];', ...
  '  }', ...
  '}', ...
  });
```

IV.C. On the usage of complex arithmetic

Unfortunately, at present time I could not manage to program any computation which makes use of complex arithmetic in a Mex file (due to my bad knowledge of C++).

20 Due to the present implementation of Snippet, this is strongly discouraged with C snippets.
21 In C++ there is no more need to invert the order of index: Matlab C++ API does it seamlessly.

V. Snippet does not replace knowledge of Matlab API

As a conclusion, it is important to stress that Snippet can help you to write small (or even not so small) Mex functions, but in order to avoid pitfalls or common errors it is mandatory to have a good knowledge of Matlab API.