excentury Documentation

Release 0.2

Manuel Lopez

August 22, 2014

Excentury is a collection of libraries written in several languages to allow the integration of C++ into scription languages such as Python and MATLAB.

CHAPTER 1

Basic Usage

1.1 What is excentury?

Excentury is a collection of libraries written in several languages to enable to the easy integration of C++ to scripting languages. By using the excentury formats we can use or create new C++ code and adapt it to computational languages such as Python and MATLAB.

1.1.1 Motivation

Scripting languages give us many advantages: faster development, extensive libraries and an overall ease of use. They are great tools and can help individuals with no programming experience to get started learning how to provide instructions to a machine. They allow us to explore ideas relatively quick without spending too much time dealing with compiler errors and many other problems that arise from low level languages.

The main disadvantage is that their execution is slow compared to the execution done by those compiled to machine code. Many scripting languages offer support to adapt low level code, thus allowing you to gain speed in your scripts. Learning how to do this is usually no easy task since it requires the user to be familiar with the low level language and the process to create the library is tedious.

To see how Excentury can help us write adaptable C++ code we present present a simple programming example.

1.1.2 Newton's Method

Newton's method is an iterative algorithm which approximates the roots of a function f by providing an initial guess x_0 and computing

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

until an accurate value is reached. To estimate the square root of a non-negative real number a we can apply Newton's method to the function $f(x) = x^2 - a$.

The following file `square-root.cpp` is a program specifically tailored to compute the square root of 5 using Newton's method.

```
#include <cstdlib>
#include <cstdio>

int main() {
    // INPUTS
6      double a = 5;
7      double x0 = 1;
8      int iter = 10;
9  
10     // ARGUMENT CHECKING
11     if (a < 0) {
12         printf("input `a` must be non-negative\n");
13         exit(1);
14     }
15 
16     // NEWTON'S METHOD
17     double x = x0;
18     for (int i=0; i < iter; ++i) {
19         x = x - (x*x - a)/(2.0*x);
20     }
21 
22     // OUTPUT
23     printf("x = %f\n", x);
24 }
```

A drawback of creating a C++ program is that passing inputs can become a tedious task. As beginners, we usually edit the program, compile and execute the program many times.

```
macbook-pro:~ jmlopez$ g++ square-root.cpp -o square-root
macbook-pro:~ jmlopez$ ./square-root
x = 2.236068
```

After changing the value of `a` to `2` we obtain

```
macbook-pro:~ jmlopez$ g++ square-root.cpp -o square-root
macbook-pro:~ jmlopez$ ./square-root
x = 1.414214
```

To avoid recompiling a program over and over again we can provide inputs via the command line or we can make the program read from a file. Consider the following modification in lines 6 through 8:

```
1  #include <cstdlib>
2  #include <cstdio>
3  
4  int main(int argc, char** argv) {
5      // INPUTS
6      double a = atof(argv[1]);
7      double x0 = atof(argv[2]);
8      int iter = atoi(argv[3]);
9  
10     // ARGUMENT CHECKING
11     if (a < 0) {
12         printf("input `a` must be non-negative\n");
13         exit(1);
14     }
15 
16     // NEWTON'S METHOD
17     double x = x0;
18     for (int i=0; i < iter; ++i) {
19         x = x - (x*x - a)/(2.0*x);
20     }
21 
22     // OUTPUT
23     printf("x = %f\n", x);
}
```

This program now accepts inputs from the command line.

```
macbook-pro:~ jmlopez$ g++ square-root.cpp -o square-root
macbook-pro:~ jmlopez$ ./square-root 5 1 10
x = 2.236068
macbook-pro:~ jmlopez$ ./square-root 2 1 10
x = 1.414214
macbook-pro:~ jmlopez$ ./square-root 10 1 10
x = 3.162278
```

This method is perfectly fine even with programs with a large amount of inputs. The problem is that the program needs to be written carefully to take into account these inputs. Notice that the program would halt execution or something would go extremely wrong if we do not provide the correct inputs to the functions.

```
macbook-pro:~ jmlopez$ ./square-root 10 1
Segmentation fault: 11
```

As it so happens, we forget the usage of a software after a time of inactivity. For this we have the "*man*" pages or some source of documentation. Documentation is often one of the most neglected parts of a software. The aim of Excentury is to keep a well documented source code which is easy to adapt to scripting languages.

1.1.3 Excentury

We have seen how a simple C++ program can be written and how troublesome making a simple routine work can be. Scripting languages function give us a safe sandbox in which we can call a function without having the program crash. Instead they throw errors which can then be dealt with. The idea behind excentury is to create one document that ties documentation, along with the source code to provide a package which can easily be exported to a scripting language of our choice.

In the next section we discuss the installation process of Excentury and then proceed to follow up on how to adapt our routine to C++, MATLAB and Python.

1.2 Installing Excentury

Before we can get started adapting our C++ code to our favorite scripting language we must have a C++ compiler installed and a copy of Excentury. First we start with the installation of Excentury.

1.2.1 Pip or Manual Installation

The easiest way to install Excentury is to use `pip`. If you wish to perform a global installation and you have admin rights then do

```
sudo pip install excentury
```

or to install in some directory under your user account

```
pip install --user excentury
```

Or if you prefer to do do a manual installation then you may do the following from the command line (where x.y is the version number):

```
wget https://pypi.python.org/packages/source/e/excentury/excentury-x.y.tar.gz
tar xvzf excentury-x.y.tar.gz
cd excentury-x.y/
sudo python setup.py install
```

The last command can be replaced by `python setup.py install --user`. See PyPI for all available versions.

1.2.2 Excentury executable

To be able to call `excentury` from the command line you must have the executable directory in your `$PATH`. This can be taken care of my calling the `install` command in `excentury`. Since the executable is not yet available you will have to call the excentury script from python.

```
python -m excentury install
```

To verify that `excentury` is now in your path you can try the help option

```
excentury -h
```

The `install` command also takes care of the C and C++ include paths. This will make sure that you can access the C++ libraries as well as the MATLAB libraries.

1.2.3 OS X

To be able to use excentury we need a C++ compiler. We may obtain this by installing XCode.

The next step is not required but if you are having trouble installing python packages then you may want to try Homebrew. Try installing it and then try installing a fresh installation of python.

1.2.4 MATLAB

Regardless of what operating system we are using, we need to make sure that our `$PATH` contains the `mex` script that comes with MATLAB.

Before we can use excentury we need to make sure that `mex` is working properly. To do a test, you should try to work with one of the mex examples provided by MathWorks.

Note: With every release of OS X and MATLAB there are a few changes that need to be done. If either the operating system or MATLAB is updated you should always first try to compile one of their examples to make sure that mex files can be compiled successfully before attempting to figure out what is wrong with excentury.

Warning: If you have OS X 10.9 and you are having trouble compiling the mex example then you may want to look at this stackoverflow question. Note that one solution is to upgrade your `gcc/g++` compilers using either homebrew or macports and specify this in the mex setup.

1.3 Getting Started

Excentury aims to provide simple and intuitive tools which we can use to develop packages with easy. This is done via a file format where we can write C++ code and give accessibility from MATLAB and Python at the same time.

The format goes roughly as follows:

```
"""Package Name

Package documentation.
"""
/* Package preamble contents */
----------------------------------------------------------------------
/* Function preamble contents */
@def{function name}
    """Function explanation. """
    @param{type1, var1, "var1 explanation"}
    @param{type2, var2, "var2 explanation"}
    @param{type3, var3, "var3 explanation"}
@body[[
    /* Function body */
]]
@ret[[
    @ret{ans1, "ans1"}
    @ret{ans2, "ans2"}
]]
/* Function epilog contents */
----------------------------------------------------------------------
/* Package epilog (if no more functions are defined) */
```

Consider the excentury file `sample.xcpp`.

```
"""Sample

This package provides the functions to compute the square root of a
non-negative real number.

"""
#include <excentury/excentury.h>

----------------------------------------------------------------------
@def{square_root}
    """Compute the square root of a number using Newton's method."""
    @param{double, a(2), "the input to the square root function"}
    @param{double, x0(1), "initial guess"}
    @param{int, iter(10), "number of iterations"}
@body[[
    if (a < 0) {
        excentury::error("input `a` must be non-negative");
    }
    double x = x0;
    for (int i=0; i < iter; ++i) {
        x = x - (x*x - a)/(2.0*x);
    }
]]
@ret[[
    @ret{x, "x"}
]]
----------------------------------------------------------------------
@def{cpp_sqrt}
    """Call the sqrt function provided by c++"""
    @param{double, a(2), "the input sqrt"}
@body[[
    double x = sqrt(a);
]]
@ret[[
    @ret{x, "x"}
]]
----------------------------------------------------------------------
```

A key difference as opposed to a regular C++ file is how we are now defining the section for the inputs and the outputs while leaving the body of the function almost intact (except for line 17 where we check for errors).

1.3.1 CPP

To try to run this example in cpp we can try the following:

```
excentury sample.xcpp to cpp
```

This will create two valid cpp files which will then be compiled into binaries which we can call.:

```
$ sample-square_root.run -h
usage: sample-square_root.run [-h] [-i] XC_CONTENT

program: sample-square_root

description:
    Compute the square root of a number using Newton's method.

parameters:
    `a`: the input to the square root function
    `x0`: initial guess
    `iter`: number of iterations

examples:

    generate an input file: sample-square_root.run -i > input_file.xc
    use the file: sample-square_root.run "`< input_file.xc`"
```

The help menu is important because it tells us how we can provide the inputs to the program. In this case we can generate an input file:

```
$ sample-square_root.run -i > input_file.xc
```

Since the xcpp file declared default values we can leave the file as is and run it as follows:

```
$ sample-square_root.run "`< input_file.xc`"
0 1
x R 8 1.414214
```

From here on we can simply modify the contents declared in "input_file.xc" to change the parameters to the function. At this moment we do not expect you to know what the xc file extension is formatted. In future sections we will go into detail on this topic since most of the development of C++ code should be done in a simple C++ file instead of MATLAB or Python. Only once the C++ code runs as expected then we can move on to using it in the interpreters.

1.3.2 Python

To be able to use our functions in the sample package we can tell excentury to give us a python package:

```
$ excentury sample.xcpp to python
```

Once excentury is done creating the necessary files we can work within python:

```
>>> import sample
>>> sample.square_root(2, 1, 10)
1.41421
>>> sample.square_root(5, 1, 10)
2.23607
>>> sample.square_root(-1, 1, 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/jmlopez/Library/Python/2.7/lib/excentury/python/sample.py", line 45, in square_root
    raise RuntimeError(xc_error_msg)
RuntimeError: input `a` must be non-negative
```

If you use the `help` function on the `sample` module you can see that there exists two functions: the one called above and `cpp_sqrt`.

```
>>> sample.cpp_sqrt(2)
1.41421
>>> sample.cpp_sqrt(5)
2.23607
>>> sample.cpp_sqrt(-1)
nan
```

1.3.3 MATLAB

Two obtain our mex function we can execute the following:

> $ excentury sample.xcpp to matlab

Then in the MATLAB prompt we can do

```
>> help sample.square_root
  sample.SQUARE_ROOT generated on Wed Aug 20, 2014 09:11:18 PM by xcpp

      Compute the square root of a number using Newton's method.

      parameters:

        `a`: the input to the square root function
        `x0`: initial guess
        `iter`: number of iterations


>> sample.square_root(2, 1, 10)

ans =

    1.4142

>> sample.square_root(5, 1, 10)

ans =

    2.2361

>> sample.square_root(-1, 1, 10)
Error using square_root_mex
input `a` must be non-negative
```

```
Error in sample.square_root (line 18)
    [~, out_str] = sample.square_root_mex(len_in, in_str);
```

Similarly, we can use the C++ function square root

```
>> help sample.cpp_sqrt
  sample.CPP_SQRT generated on Wed Aug 20, 2014 09:11:20 PM by xcpp

      Call the sqrt function provided by c++

      parameters:

        `a`: the input sqrt


>> sample.cpp_sqrt(2)

ans =

    1.4142

>> sample.cpp_sqrt(5)

ans =

    2.2361

>> sample.cpp_sqrt(-1)

ans =

   NaN
```

1.3.4 What's Next?

You may use this example to try to experiment creating function which can be called from CPP, MATLAB or Python. The Excentury documentation is still far from complete, for the moment you can look over the source code to see if there are any functions that may be of interest and give them a try.

1.4 Excentury Format

To be able to seamlessly adapt a piece of C++ code to a script language we need a way to communicate between C++ and the scripting language. To understand how this communication process works we must first examine the excentury file format.

1.4.1 Basic Types

In the C++ language there are several basic datatypes. These types help us represent integers, real numbers and characters. To be able to store information we need to be able to store the type of the object we are storing along with its value. This could have been done in several ways but for simplicity we have chosen to declare a datatype by the type of object that is being stored along with the number of bytes that it is required for it in memory.

There are several datatypes which represent integers, these are `signed char`, `short int`, `int` and `long int`. To be able to represent `short int` for instance we can state `I2`, meaning an integer of 2 bytes. The following table shows all the basic datatypes in C++ along with its excentury representation.

Type name	Excentury	Denotes
char	`C 1`	character of 1 byte
unsigned **char**	`N 1`	natural number of 1 byte: 0 to 255
unsigned short **int**	`N 2`	natural number of 2 bytes
unsigned **int**	`N 4`	natural number of 4 bytes
unsigned long **int**	`N 8`	natural number of 8 bytes
signed **char**	`I 1`	integer of 1 byte: -128 to 127
short **int**	`I 2`	integer of 2 bytes
int	`I 4`	integer of 4 bytes
long **int**	`I 8`	integer of 8 bytes
float	`R 4`	real number of 4 bytes
double	`R 8`	real number of 8 bytes

These basic types are the building blocks for all possible datatypes that we might need, these help us build structures which need to be adapted to excentury in order to store them in a file.

1.4.2 XC files

The idea behind the excentury format is that the content of the file must contain all the information so that we may load its data into a scripting language. The following file for instance, does not contain all the necessary information to load variables into a scripting language.

```
-1 3.14
```

Here we can tell that the file contains two values, the integer -1 and the real number 3.14. This however, does not say how it was previously stored in C++. One way to correct this is to write the file as follows

```
2
a I 4 -1
b R 8 3.14
```

The file states that it contains two variables. The first one is an integer of 4 bytes with value -1 and it should be assigned the name `a` when loaded. The second variable is a real number of 8 bytes of value 3.14 and should be named `b`.

1.4.3 Structures

To store structures we need to find a way of serializing the object with the minimum amount of information. Suppose that we wish to store two structures, a `Point` and a `Line`.

```
class Point {
public:
    double x, y;
    Point(): x(1), y(1){}
    Point(double a1, double b2): x(a1), y(b2){}
};

class Line {
public:
    Point a, b;
    Line(): a(0, 0), b(1, 1) {}
    Line(int a1, double b1, int a2, double b2):
        a(a1, b1), b(a2, b2) {}
};
```

For the moment, let us assume that we have taught excentury how these two structures need to be serialized. A file containing a `Point` named `point_obj` and a `Line` named `line_obj` may possibly look as follows

```
2
Point x R 8 y R 8
Line a S Point b S Point
2
point_obj S Point 100.0 200.0
line_obj S Line 1.0 2.0 3.0 4.0
```

This file states that there are 2 structure definitions. The first one is for `Point`. To read a definition we simply read pairs of tokens: name and type. For a `Point` we have that its first member is `x` and it is a real number of 8 bytes (`R 8`). The second member is `y` and it is a real number of 8 bytes. Here we have to rely on the new line character to know that there are no other members for `Point`. Similarly, for the `Line` definition we have that its first member is named `a` and it is a `Point` structure (`S Point`) while its second member is a `Point` structure named `b`. This first section we just described is the dictionary for the file. This part contains all the definitions of structures stored in the file.

The second part contains the actual data. Here we can see that the file contains 2 objects. The first one is called `point_obj`. This is a structure of type `Point`. Since we now know the definition for a `Point` we now know that we expect two values: a real number `x` and a real number `y`. In this case these values are 100 and 200. The second object we have a `Line` structure. This one is made up of two points. So we must first the first point `a` which has members `x` and `y`, thus the member `a` has member `x` of value 1 and member `y` of value 2. Similarly for the member `b` we have values 3 and 4.

To store a structure we first need to tell Excentury how to store it. This however, will be covered in a later section. For now, we must mention one last object that is essential to the excentury format.

1.4.4 Tensors: Multidimensional Arrays

Arrays are essential to every programming language. Here we will give a brief introduction on how we decided to store them in the excentury format.

To store an array of integers of 4 bytes we could state the variable name followed by `A I 4` followed by the number of elements in the array and their values. For instance

```
array_name A I 4 3 1 2 3
```

This was the original idea on how to store arrays. Similarly for matrices we would use the letter `M` but this time we would use two values to store its dimension.

```
matrix_name M I 4 2 3 1 2 3 4 5 6
```

One problem with this notation is that we assumed that the information was stored in column major form. That is the matrix in the previous file is

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

We can overcome this problem by adding a 0 if we want column major or 1 if we want row major. The main problem when storing arrays and matrices however is that if we continue naming these structures we will soon find that we run out of names. For instance, an array is simply a sequence of objects. An array of arrays is called a matrix. An array of matrices is, well, a tensor of dimension 3. A multidimensional array is a tensor.

To specify a tensor we can use `T <type>` where type is either a basic type, a structure or a tensor. After that we must specify if it is row major or column major followed by the number of dimensions of the tensor and its dimensions. Finally we write the data.

To specify the previous array and matrix we would write the following excentury file

```
0 2
array_name T I 4 0 1 3 1 2 3
matrix_name T I 4 0 2 2 3 1 2 3 4 5 6
```

There is one type of array which is special that treating it as a tensor may be considered a waste of resources. A sequence of characters is usually known as a `string`. This type of array is special in excentury and it has been given the type `W`. For instance to store the string `"hello world"` we can use:

```
0 1
str_obj W 11 hello world
```

This says that `str_obj` is a word (`W`) of 11 characters.

1.4.5 Summary

The excentury file format takes the following form

```
<number of definition>
<structure name> [<member name> <type>] ...
...
<number of objects>
<variable name> <type> <data>
<variable name> T <type> <row major:1, column major: 0> <dimension> <dimensions> <data>
<variable name> W <string size> <data>
...
```

To write this file format we can use C++, Python or MATLAB. See each of their sections for more information and examples on how to do it.

1.5 Armadillo

Excentury supports the use of armadillos datatypes. Armadillo is a C++ linear algebra library. See more information on the library at http://arma.sourceforge.net/.

To see an example see the following directory

> https://github.com/jmlopez-rod/excentury/tree/master/tests

There we can find `xcpp/arma.xcpp` which contains an example. We also need to consider the configuration file `xcpp.config` since it contains information on how to compile excentury files using the armadillo library.

CHAPTER 2

C++

2.1 Debugging

Writing C++ routines is no easy task. Many times a program ends abrutly with an error. A common error is a "*segmentation fault*" error. When a program displays this message it usually means that the program was trying to access a memory location outside its address space. Many other errors may occurr but finding these errors is usually a time consuming task and several debugging techniques are needed to find them.

A common debugging technique is to print messages on your program so that you may debug or trace the execution of the program. Excentury provides three macro functions to aid in future debugging tasks. These functions expand to a command if the `DEBUG` macro is defined before the inclusion of `excentury.h`. There are three values that the `DEBUG` macro can take. These three values are the several levels of debugging which can facilitate the debugging task at hand and in the future.

To explore what each of these levels do we will consider the file `example.cpp` which looks as follows

```cpp
#include <excentury/excentury.h>

int main() {
    debug("This message is only seen with DEBUG set to 2\n");
    trace("This message is only seen with DEBUG set to 3\n");
    printf("Hello world\n");
    exitif(true, NULL, "This is a test to check that DEBUG is on.\n");
    printf("Debug was turned off...\n");
}
```

Notice that a compilation without defining the `DEBUG` macro will result in the following

```
$ g++ example.cpp -o example.run
$ ./example.run
Hello world
Debug was turned off...
```

This is because `debug`, `trace` and `exitif` expanded to nothing, *i.e.*, `example.cpp` expanded to

```cpp
#include <excentury/excentury.h>

int main() {
    printf("Hello world\n");
    printf("Debug was turned off...\n");
}
```

2.1.1 Level 1: `exitif`

This is the most basic level and it will allow you to use the `exitif` function.

```
exitif(condition, function_call, ...)
```

This is a macro which expands to the following:

```
if (condition) {
    printf(...);
    function_call;
    exit(1);
}
```

To can either define `DEBUG` before inclusion of `excentury/excentury.h` or define it during the call to `g++`

```
$ g++ -DDEBUG=1 example.cpp -o example.run1
$ ./example.run1
Hello world
ERROR CAUGHT BY example.cpp line 7 executing:

    int main()

The error occurred because:  true

This is a test to check that DEBUG is on.
```

In this case, notice that since `exitif` was defined and the condition to exit was satisfied (any statement that evalutes to `true`) the program printed a statement explaining the error detected and halted the execution of the program.

2.1.2 Level 2: `debug`

This level provides the function `debug`. These messages should only be used while debuging. If the messages could help in the future then we should replace `debug` with the level 3 function `trace`. The `debug` function behaves as `printf` but will only be expanded in levels 2 and 3.

```
$ g++ -DDEBUG=2 example.cpp -o example.run2
$ ./example.run2
This message is only seen with DEBUG set to 2
Hello world
ERROR CAUGHT BY example.cpp line 7 executing:

    int main()

The error occurred because:  true

This is a test to check that DEBUG is on.
```

Notice the message from level 1 was displayed as well as the message provided to the `debug` function. Again, these messages are meant to be temporary in order to find out what is going on. We could have used the `printf` function but soon we will not be able to differentiate between the actual statements that we want to display and those that were temporary debugging messages.

2.1.3 Level 3: `trace`

Provides the function `trace` which will display messages when level 3 is active. These messages are meant to be permanent messages and should be designed to help the user have a better idea of what is going on with the program.

```
$ g++ -DDEBUG=3 example.cpp -o example.run3
$ ./example.run3
This message is only seen with DEBUG set to 2
This message is only seen with DEBUG set to 3
Hello world
ERROR CAUGHT BY example.cpp line 7 executing:

    int main()

The error occurred because:  true

This is a test to check that DEBUG is on.
```

The debugging level 3 is really meant to be used as a last resort tool. If done correctly, the debugging level 1 should catch the errors. If this is not enough then we may display messages by using level 2. If all fails then we may want to activate the `trace` messages to see if there are any useful messages.

This version of the documentation was built August 22, 2014.