

# CS107/AC207 Final Project: Milestone 2

*Matthew Hawes, Junyi Guo, Jack Scudder, Paul Tembo, Arthur Young*

Due: 11:59 PM, November 19, 2020

## 1 INTRODUCTION

CMAutoDiff is a library that computes derivatives in the forward mode. In computational engineering and data science, optimization problems are at the core of every challenge that individuals and teams in the field face. Finding derivatives is the common approach in dealing with optimization problems and sometimes it is hard and time-consuming to calculate the symbolic derivative of real-life equations. Automatic differentiation gives us the ease of solving the derivative of complex functions with accuracy to machine precision. In contrast with the difficulties in implementing and dealing with complex derivatives using symbolic differentiation and numerical differentiation, automatic differentiation provide us a more precise, efficient, and scalable way to compute derivatives when equations increase in complexity and number of inputs variables. In its final release, this library will also contain a module to perform the reverse mode of automatic differentiation.

## 2 BACKGROUND

The basic principle of automatic differentiation can be understood as a systematic evaluation of the chain rule of differentiation. Let  $x \in \mathbb{R}$ ,  $y \in \mathbb{R}$  be mapped via the composite function  $f : x \rightarrow y$  where  $f$  is defined by a sequence of evaluations of sub-functions as follows:

$$f := f_n(\dots(f_3(f_2(f_1(x)))))) = y \quad (1)$$

the chain rule of differentiation gives us that:

$$\frac{dy}{dx} = \frac{d}{dx}(f) = \prod_{i=1}^n \frac{d}{d\omega_i}(f_i(\omega_i)) \quad (2)$$

where we take turns evaluating the derivative of each function  $f_i(\omega_i)$ . Thus, we can consider each function  $f_i$  to be a mapping from  $\omega_i \rightarrow \omega_{i+1}$

Importantly, automatic differentiation does not yield an analytical expression for the derivative of a given function. Rather, if one considers the process of automatic differentiation in a black box format, then there are two inputs to automatic differentiation, the function form and the evaluation point. Consider a composite function  $f : f_n(\dots(f_3(f_2(f_1(\omega_1))))))$  defined over the differential field  $\Omega$ . We seek the value of  $\frac{df}{dx}|_{x_0}$ , where  $x_0 \in \Omega$ . We can view automatic differentiation as an operator  $\mathcal{D} : f, x_0 \rightarrow \frac{df}{dx}|_{x_0}$ .

There are two approaches one can take to apply the chain rule given by equation 2, and they are referred to by the terms *forward* and *reverse* accumulation. In forward accumulation, the evaluation of the derivative of composite function  $f$  is considered recursively from “inside out,” meaning:

$$\begin{aligned}
 \mathcal{D}_{x_0}\{f\} &:= \mathcal{D}_{x_0}\{f_{n-1}\} \frac{\partial f_n}{\partial \omega_n} \Big|_{f_{n-1}(\omega_{n-1})} \\
 &= \mathcal{D}_{x_0}\{f_{n-2}\} \left( \frac{\partial f_{n-1}}{\partial \omega_{n-1}} \Big|_{f_{n-2}(\omega_{n-2})} \right) \left( \frac{\partial f_n}{\partial \omega_n} \Big|_{f_{n-1}(\omega_{n-1})} \right) \\
 &\vdots \\
 &= \frac{\partial f_1}{\partial \omega_1} \Big|_{x_0} \left( \prod_{i=2}^n \frac{\partial f_i}{\partial \omega_i} \Big|_{f_{i-1}(\omega_{i-1})} \right)
 \end{aligned}$$

This implies a recursion where one starts at the lowest embedded sub function  $f_1$  and systematically evaluates the expression outwards. The process would therefore be:

1.  $\omega_{it} \leftarrow x_0$
2.  $\dot{x} \leftarrow 1$
3. for  $i$  in range( $n$ )
 

$\omega_{it} \leftarrow f_i(\omega_{it})$   
 $\dot{x} \leftarrow \dot{x} \cdot \frac{\partial f_i}{\partial \omega_i} \Big|_{\omega_{it}}$

Contrast the above workflow with that of *reverse* accumulation: given the standard chain rule expression,

$$\begin{aligned}
 \frac{\partial f}{\partial x} \Big|_{x_0} &= \left( \frac{\partial f_n}{\partial \omega_1} \Big|_{f_1(\omega_1)} \right) \left( \frac{\partial \omega_1}{\partial x} \Big|_{x_0} \right) = \left( \frac{\partial f_n}{\partial \omega_2} \frac{\partial \omega_2}{\partial \omega_1} \right) \Big|_{f_1(\omega_1)} \left( \frac{\partial \omega_1}{\partial x} \right) \Big|_{x_0} = \dots \\
 &= \left( \prod_{i=n}^2 \frac{\partial \omega_i}{\partial \omega_{i-1}} \Big|_{f_{i-1}(\omega_{i-1})} \right) \left( \frac{\partial \omega_1}{\partial x} \Big|_{x_0} \right)
 \end{aligned}$$

We note that the above implies a different order of operations. Instead of beginning with the innermost component of the composite function, we are instead differentiating the outermost expression with respect to the next outermost function, and then carrying until we reach the independent variable  $x$ .

### 3 HOW TO USE

#### 3.1 Package Interaction

The following are Guidelines for download, installation, and use of package via git clone. The package is available for download on **GitHub** through the following **URL**:

`https://github.com/Cache-Money404/cs107-FinalProject.git`

It can be cloned from the command line as follows:

```
$ git clone 'https://github.com/Cache-Money404/cs107-FinalProject.git'
```

Figure 1: Clone the package using git clone

The package can also be downloaded (see image below), to do so :

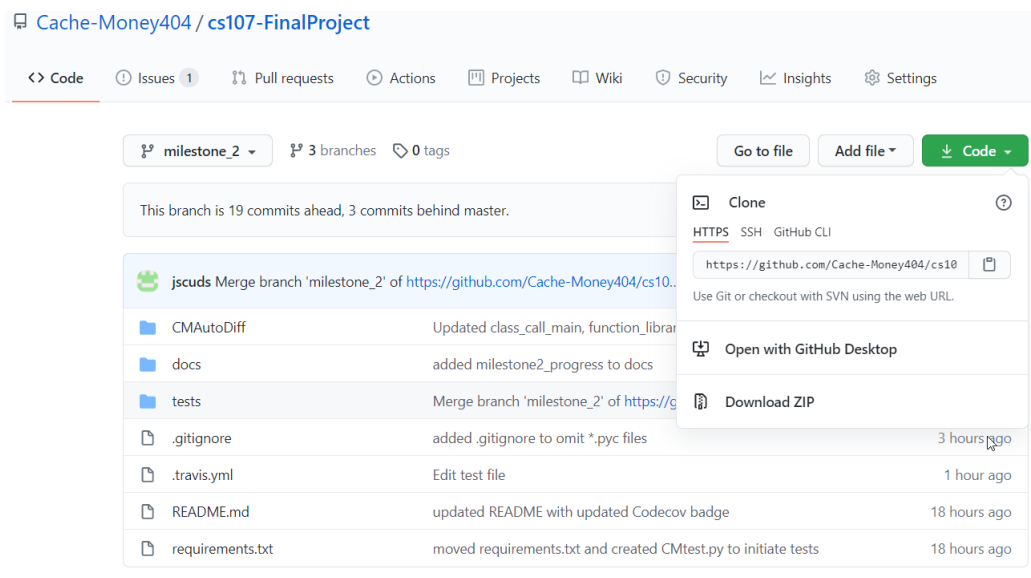


Figure 2: Download Package

1. Click on the "Code" Tab
2. Scroll down and download the ZIP file.
3. UnZip the files into your work directory and install using the package installation in the next section.

#### 3.2 Package Framework

The package doesn't use any formal framework at this point (in the future it will use PyPi). It can be installed using the **git clone and download** instructions listed above.

1. To install dependencies for the package in Python 3, run the following which is in the main directory of the library:

```
pip install -r requirements.txt
```

2. The library can be imported and used from the path on your local system from which you installed it.

### 3.3 Imports

To ensure that CMAutoDiff works correctly, ensure you import the following at the beginning of your .py file or the environment in which you wish to use CMAutoDiff:

```
import numpy as np
from function_library import function_library
from CMOBJECT import CMOBJECT
from FuncObj import FuncObj
```

### 3.4 User Interface

After importing the necessary modules, a user creates a class instance of an CMOBJECT with the value to be used as input to a function.

#### Steps for Instantiating Variables and Functions

1. Initialize an input variable (i.e. 'x') with the value at which the function will be evaluated.

---

```
x = CMOBJECT(3)
```

---

2. Declare a function (i.e. 'f') with the variable and the FuncObj class.

*You can decide between 'sin', 'cos', 'tan', 'log', and 'exp'*

Here is a simple example:

$$f(x) = \tan(x)$$

---

```
f = FuncObj( 'tan ', x1 )
```

---

Here is a more complex example:

$$f(x) = \sin(\tan(x)) + 2^{\cos(x)} + \sin(x)^{\tan(x)e^x} - \cos^2(x)$$

---

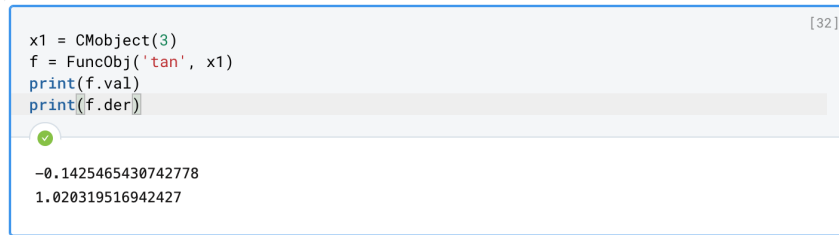
```
f = FuncObj( 'sin ', FuncObj( 'tan ', x1 ))
    + 2**(FuncObj( 'cos ', x1 ))
    + FuncObj( 'sin ', x1 )**(FuncObj( 'tan ', x1 ))**(FuncObj( 'exp ', x1 ))
    - FuncObj( 'cos ', x1 )**2
```

---

3. `f.val` will return the value of the function evaluated at the specific value
4. `f.der` will return the derivative at the specific value

In the final product, we plan on using `flask` to create a GUI that will run the program, receive all of the inputs, and display the output values and graphs.

### Example Screenshots:

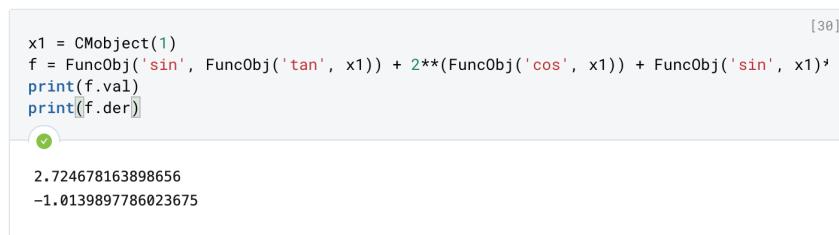


```
x1 = CMOBJECT(3)
f = FuncObj('tan', x1)
print(f.val)
print(f.der)
```

[-0.1425465430742778  
1.020319516942427]

This screenshot shows a Jupyter Notebook cell with index [32]. The code defines a CMOBJECT with value 3, creates a FuncObj for the 'tan' function, and prints its value and derivative. The output shows the value is approximately -0.1425 and the derivative is approximately 1.0203.

Figure 3



```
x1 = CMOBJECT(1)
f = FuncObj('sin', FuncObj('tan', x1)) + 2*(FuncObj('cos', x1)) + FuncObj('sin', x1)*
print(f.val)
print(f.der)
```

[2.724678163898656  
-1.0139897786023675]

This screenshot shows a Jupyter Notebook cell with index [30]. The code defines a CMOBJECT with value 1, creates a complex FuncObj combining 'sin', 'tan', and 'cos' functions, and prints its value and derivative. The output shows the value is approximately 2.7247 and the derivative is approximately -1.0140.

Figure 4

## 4 SOFTWARE ORGANIZATION

### 4.1 Directory Structure

```

main
├── tests
│   └── directories for containing future tests
├── docs
│   ├── design document
│   └── funcs.txt with all functions supported by this application
├── CMAutoDiff
│   ├── CObject.py
│   │   └── main AD object creation
│   ├── FuncObj.py
│   │   └── helper class for performing the elementary functions such as sine and cosine
│   ├── function_library.py
│   │   └── helper functions for calculating the function values as well as derivatives
│   │       for the elementary functions such as sine and cosine
│   ├── test_CM.py
│   │   └── file that contains test cases for this milestone2 interim release
│   ├── *future inclusion* graphics construction
│   │   └── all files for constructing graphic visualization
│   ├── *future inclusion* interface
│   │   └── main script for user interface
└── *future inclusion* README.md
    └── detailed instructions on how to run the interface

```

The directory tree structure above shows our general design for our package. The main directory contains 4 sub-directories that perform automatic differentiation and subsequently return specified outputs and the forward mode graph. The detailed functionality for each file/directory is listed below:

- docs/ : contains all documentation for using, designing, and developing the CMAutoDiff package
- CMAutoDiff/ : contains all the code for the user interface, the common math functions, and a basic test file
- tests/ : directory for future use that will contain all test files for CMAutoDiff

## 4.2 Modules

The following modules are included in the `CMAutoDiff` package:

- `CObject.py` : contains the `CObject` that a user incorporates into a function to perform automatic differentiation; overloads arithmetic operations to calculate values and derivatives of a function.
- `function_library.py` : contains elementary functions and their derivatives. At this time the user can calculate sine, cosine, tangent, exponent (Euler's number), and natural logarithm (aka. `log` or `ln`). All math functions make use of the `numpy` library.
- `FuncObj.py` : contains the `FuncObj` that allows a user to calculate elementary values and derivatives of a `CObject`. Inherits from the `CObject` class.
- `test_CM.py` : contains developer tests for the package.

## 4.3 Test Suite

Currently, the test suite consists of:

- TravisCI and CodeCov, utilized with their badges on the `README.md` in the main directory.
- A developer series of tests at `CMAutoDiff/test_CM.py` that test every function and object method

In our final release, the developer tests will be included in the sub-directory `tests/`. Examples of program tests and handled cases currently included in the test suite:

- Simple differentiation of 1 input and 1 function
- Complex differentiation of 1 input variable and 1 output function of several nested trigonometric, logarithmic, and power calculations (tests every basic function coded in the library eg. trigonometric: sine, cosine, tangent; logarithmic: exp, ln)
- Newton's Method as a root-finding algorithm
- Every overloaded class method for the `CObject` class
- Whether or not the function is differentiable using common derivative rules.

## 4.4 Package Distribution

Currently, a user will follow the instructions in 3.1 Package Interaction to install and use `CMAutoDiff`. In the final release (12 December 2020), our package will be accessible via PyPi/PIP. A user will install the latest version of `CMAutoDiff`, use `pip install` as follows (using ssh to avoid repetitively inputting credentials):

```
pip install git+https://github.com/Cache-Money404/cs107-FinalProject.git
```

## 4.5 Package Framework

The package doesn't use any formal framework and in its final release (12 December 2020) it will be installed using the `pip` instructions listed above.

## 5 (CURRENT) IMPLEMENTATION

As it stands, the CMAutodiff algorithm is in a proof of concept state that successfully performs forward mode automatic differentiation for a limited set of trigonometric and analytical functions. A user looking to use our current framework installs the library via the instructions detailed in section 3.1 and section 4.4, and will notice that the `CMAutoDiff/` directory contains a variety of python scripts that form the instructions of the algorithm. Of particular note is the script `test_CM.py`, which offers a variety of test cases for our proof of concept implementation, and we will discuss below to highlight the workings of our current framework.

Let us suppose that the user provides the following function

$$f(x) = \sin(\tan x) + 2^{\cos(x)} + (\sin(x))^{\tan(x)e^x} - (\cos(x))^2 \quad (3)$$

The above function is featured in the `test_difficult_derivative_case()` function, and is differentiated at a value of  $x = 1$ :

$$\left. \frac{d}{dx} f(x) \right|_{x=1} \quad (4)$$

To do so, we initialize a `CMObject` to represent the variable  $x = 1$ , by defining `x1 = CMObject(1.0)`. Then, we invoke the `FuncObj` to ascribe the properties of various supported functions into the `CMAutodiff` framework. These two objects have much in common, and in fact the `FuncObj` inherits from the `CMObject` class, but with a constructor that overrides that of `CMObject` to incorporate the chain rule of differentiation. While the `CMObject` is initialized with its variable seed and a default derivative of 1, the `FuncObj` is initialized with `string` and `arg` inputs. The former input specifies the properties of the function that the `FuncObj` is emulating, and the latter specifies the object on which the function operates. For example, in the above expression for  $f(x)$ , a term  $\cos(x)$  is encoded by the definition of a `FuncObj` with arguments `FuncObj("cos(x)", CMObject(1.0))`. Similarly, the term  $\sin(\tan(x))$  is encoded by `FuncObj("sin(x)", FuncObj("tan(x)", CMObject(1.0)))`. Thus, the implemented object structure is capable of representing complex expressions using these classes, and will perform forward mode automatic differentiation by recursively considering the constituents of the defined expression featuring instances of the `CMObject` and `FuncObj` classes, provided that the user adheres to the library of supported functions defined in the `function_library.py` script which is invoked by the `FuncObj` class for each instance.

As a proof of concept case, the function `test_newtons_method()` provides a framework for the definition of a function for which a root value is sought. Then, the method iteratively applies Newton-Raphson updates, using the specified function to recompute the value and derivative of the function at new values of  $x$ . We show that for our given test case, our automatic differentiation framework used in a Newton-Raphson algorithm finds the roots of the function  $f(x) = x^2 + \ln(x) + x$  in 5 iterations.



## 6 FUTURE FEATURES

In this milestone, we implemented the basic modules that are capable of doing forward mode of automatic differentiation, with implementation based largely on elementary differentiation rules and the chain rule. In order to further enhance our library, we plan to implement the reverse mode of automatic differentiation on top of our current implementation, which is particularly useful in completing the back propagation calculation for deep learning tasks.

The major differences between the forward mode and the reverse mode of automatic differentiation is that the previous solution starts from the inputs and calculate the derivative at every step along the way to the output, whereas the latter solution starts from the outputs and propagates back along the pathway to the inputs. However, both solutions require implementation of chain rules to complete the calculation. Thus, our previous implementation of automatic differentiation can also be useful for implementing the reverse mode. Instead of rewriting the entire library, our primary course of action is to keep the current forward mode structure, and add another module that can make use of the forward mode modules recursively – starting from the output to achieve the backward motion. To be more detailed, the new modules may require a tree structure or other data structures that can allow us to retrieve the previous calculation recursively on top of the back differentiation rules and chain rules we have in hand currently.

One possible difficulty in implementing the reverse mode of automatic differentiation is how to keep track of the previous calculations. Specifically we need to solve how to remember the parent nodes if we are going to use a tree structure for keeping track of the entire calculation. One concrete example can be:

$$f(x) = v(x) + w(x) \quad (5)$$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f}{\partial v} \frac{\partial v}{\partial x} + \frac{\partial f}{\partial w} \frac{\partial w}{\partial x} \quad (6)$$

$$(7)$$

In this case, we not only need to remember  $\frac{\partial f}{\partial v}$  and  $\frac{\partial f}{\partial w}$ , we also need to keep a record of  $\frac{\partial v}{\partial x}$  and  $\frac{\partial w}{\partial x}$  during our backward calculation. Determining a way of efficiently implementing such a structure while also making good use of the previous implementation is going to be our major focus when developing this future feature.

## 7 FIGURES FOR FINAL IMPLEMENTATION

Basic Structure (figure 1)

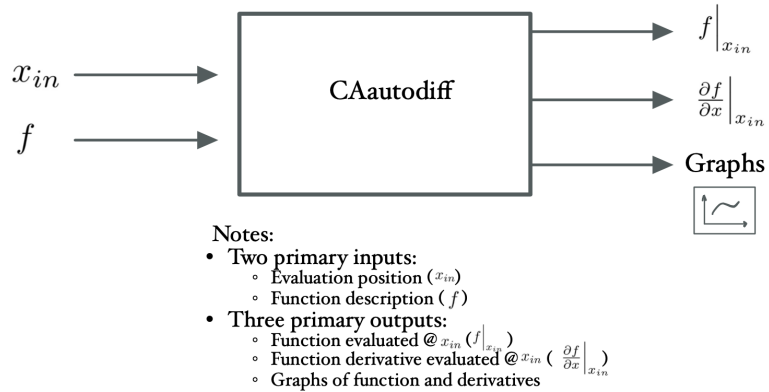


Figure 5

Internal working of the Autodiff (figure 2)

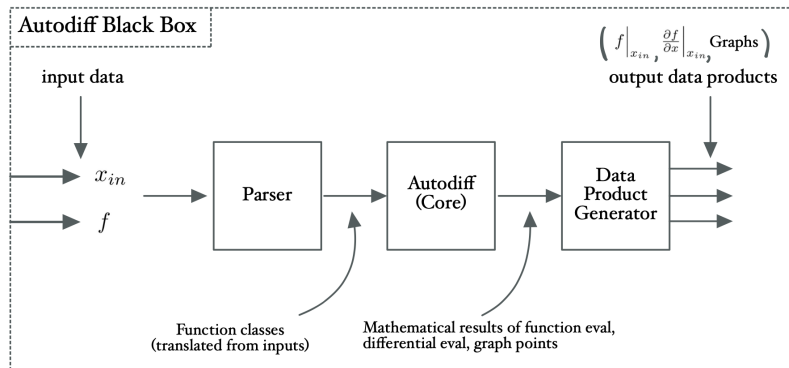
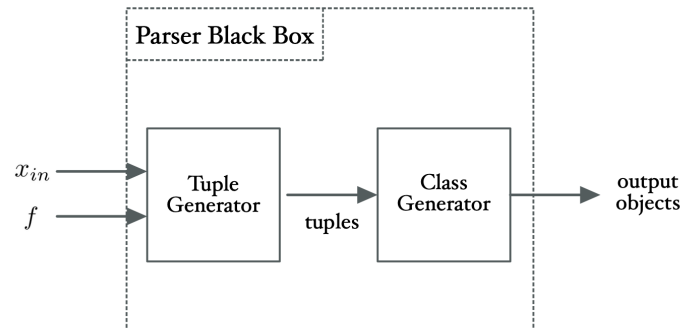


Figure 6

### Internal working of Parser (figure 3)

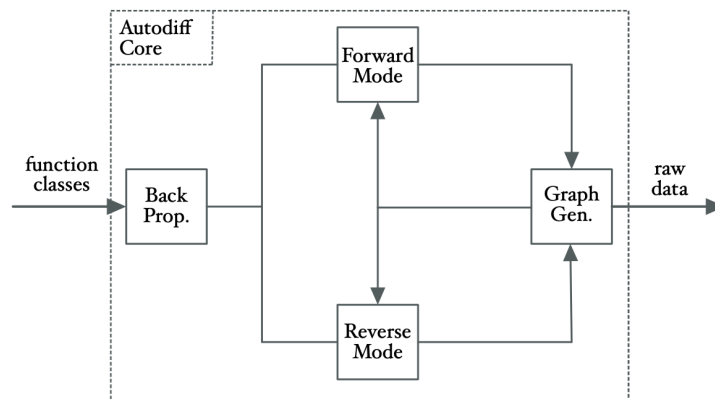


#### Notes:

- Raw inputs read in as strings.
- Strings are parsed for:
  - Arithmetic symbols
  - Function variables
  - "Special" functions
- The above symbols and functions are identified and expressed as tuples respecting order of operations.
- Tuples are used to instantiate function classes.
- Function classes have evaluation methods.

Figure 7

### Internal working of Autodiff (core) black box (figure 4)

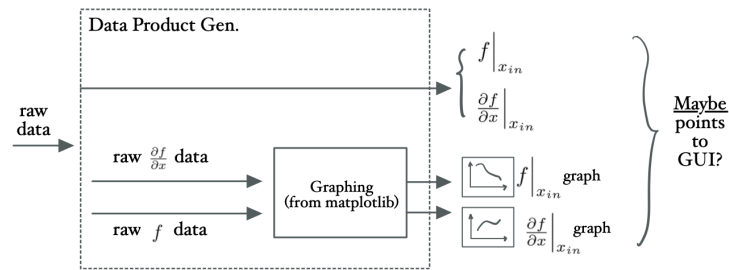


#### Notes:

- The function classes and their evaluation hierarchy are inputs.
- The special case of back propagation is considered (user input?).
- The forward or backward modes recursively performs auto-diff.
- The outputs are sent to graph generation, which loops the auto-diff until enough points to graph are generated.
- Raw data comes out.

Figure 8

### Internal working of Data Product Generator Black Box (figure 5)



#### Notes:

- Essentially, we take the raw data products from Autodiff and convey the results to the user.
- Maybe will use GUI, so this box will work with that.

Figure 9