CS107/AC207 Final Project: Milestone 1

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

Due: 11:59 PM, November 3, 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.

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 $f : x \to y$ where f is defined by a sequence of evaluations of sub-functions as follows:

$$f := f_n(...(f_3(f_2(f_1(x))))) = y \tag{1}$$

the chain rule of differentiation gives us that:

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

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 $\omega_i \to \omega_{i+1}$

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 $f : f_n(...(f_3(f_2(f_1(\omega_1)))))$ defined over the differential field Ω. We seek the value of $\frac{df}{dx}|_{x_0}$, where $x_0 \in \Omega$. We can view automatic differentiation as an operator $\mathcal{D} : f, x_0 \to \frac{df}{dx}|_{x_0}$.

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

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

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

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

To run this package, execute `CMautodiff.sh` to install appropriate dependencies and initialize the command line interface. All run-time instructions are be displayed on-screen in the command prompt. In the first stages of development, the user is be prompted to list the variables of the function, their evaluated/seed values, the number of sub-functions, and type out each sub-function in order from outer- to innermost function. Later in development, the user will only need to input a single string that will be parsed by the program. Further detail is provided in the *Implementation* section. The package outputs values and graphs of the function and its derivative evaluated at the user-specified points.

3.2 User Interface

In the first phase of development, the user executes the program and enters all inputs from the command line. Objects are instantiated based on progressive prompts printed at the command line. The prompts below are given further context in the *Implementation* section.

The user executes the program from the command line. Upon being run, the program will prompt the user to provide the position and function specification inputs. The user will provide these inputs, which will be initially read in as strings. In our initial development phase, we intend for the algorithm to prompt the user with the following questions:

1. List the variables of your function and at what values you wish them to be evaluated (e.g. x0 = 1, x1 = 2, x2 = 3, ...):
2. How many composite functions form your function (e.g. 4)?
3. Using "#" to indicate where the **next** sub-function (i.e. function 2) will go, from **outermost** to innermost, enter sub-function number 1 (e.g x + y*sin(#)):
4. Enter sub-function number 2 :
5. ...
6. Enter sub-function number [last]:

In later phases, 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.

3.3 Imports

When executed, `CMautodiff.sh` will automatically import the necessary libraries: `numpy`, `matplotlib`, and `tkinter`.

4 SOFTWARE ORGANIZATION

4.1 Directory Structure

```
main
├── tests
│   └── all test files
├── docs
│   ├── design document
│   └── funcs.txt with all functions supported by this application
├── CMautodiff.sh
│   └── bash script for users to run
├── bin
│   ├── AD algorithm
│   │   └── all files for doing AD
│   ├── math functions
│   │   └── files for defining math functions
│   ├── graphics construction
│   │   └── all files for constructing graphic visualization
│   └── interface
│       └── main script for user interface
└── 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 5 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:

- `tests/` : contains all test files for our automatic differentiation package
- `docs/` : contains all documentations for designing and developing the automatic differentiation package
- `CMautodiff.sh` : the main script for users to start the interface, contains all the commands that users need for running the interface as well as the automatic differentiation program
- `bin/` : contains all the coded for the user interface, the primary algorithm for doing automatic differentiation, the common math functions, and the graphic modules for visualize automatic differentiation
- `README.md` : contains detailed instructions on how to run the package and the interface for the users

4.2 Modules

The following modules are included:

- `CMinterface.py` : build the user interface
- `CMderiv.py` : calculates derivatives for math functions input by the users (exponential, logarithmic, trigonometric)
- `CMcalc.py` : performs automatic differentiation of complex functions in forward mode

- `CMgraph.py` : prints graph for automatic differentiation
- `CMtest.py` : contains developer tests for the package

4.3 Test Suite

The developer tests will be included in the sub-directory `tests/`. `TravisCI` and `CodeCov` are utilized with their badges on the `README.md` in the main directory.

Below are examples of program tests and handled cases included in the test suite:

- User input validation (correct function, correct input variables, correct menu selection; else returns acceptable entries and references documentation)
- Simple differentiation of 1 input and 1 function (tests every basic function coded in the library eg. trigonometric: sine, cosine, arctangent; logarithmic: exp, ln, log; etc.)
- Complex differentiation of 4 inputs and 4 functions.
- Whether or not the function is divided by zero or infinity.
- Whether or not the function is differentiable using common derivative rules.

4.4 Package Distribution

Our package is accessible via PyPi/PIP. To 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 can be installed using the `pip` instructions listed above.

5 IMPLEMENTATION

At its core, our auto differentiation program can be considered in the following black box analogy: a value and a function specification in the form of a string are inputted, and the values of the function, its derivative, and its graph are outputted (see Figure 1).

To understand how the algorithm will work, we consider the following flow from input to output (see Figure 2).

The user executes the program from the command line. Upon being run, the program will prompt the user to provide the position and function specification inputs. The user will provide these inputs, which will be initially read in as strings. In our initial development phase, we intend for the algorithm to prompt the user with the following questions:

1. List the variables of your function and at what values you wish them to be evaluated (e.g. x0 = 1, x1 = 2, x2 = 3, ...):
2. How many composite functions form your function (e.g. 4)?

3. Using "#" to indicate where the **next** sub-function (i.e. function 2) will go, from **outermost** to innermost, enter sub-function number 1 (e.g x0 + x1*sin(#)):
4. Enter sub-function number 2 :
5. ...
6. Enter sub-function number [last]:

At later stages in development, when we are satisfied with our proof of concept implementation, we intend to supplant this input framework with one in which the user inputs a single string representing the function form in a manner that a calculator would recognize, obeying standard order of operations. The program will parse the input to build the specified composite function.

The results of these inputs will be read in as strings and passed to the first major component of our algorithm, which we shall call the *input parser* (see Figure 3). In short, the input parser takes in the provided strings and uses them to build the composite function in such a way that it can be *recursively* interpreted by our program. The parser begins by reading the input variables in the first prompt and will save their associated strings and values separately. For example, for an input of "x0 = 1, x1 = 2, x2 = 3," the parser will form two separate tuples of form tuple1 $= (\text{“}x0\text{”}, \text{“}x1\text{”}, \text{“}x2\text{”})$ and tuple2 $= (1, 2, 3)$. The former tuple informs the next step of the process, the later comes in to play later.

The parser then considers the inputted sub-functions, in which the first step will be to identify any basic arithmetic symbols within the raw input string, such as `+`, `-`, `/`, `*`, `**`, `(`, `)`. The parser will consider these arithmetic operators and use them to partition the rest of the input string for further inspection. Comparing the partitioned input strings to a compiled list of acceptable strings corresponding to supported functions (respecting variations in upper vs. lower case, spaces, etc.) and variable names provided by input 1, the parser assigns a unique integer identifier corresponding to a supported mathematical function that will be used later in our program. If an inputted string does *not* have sub-strings corresponding to a recognizable function, the algorithm will prompt the user to reinput the unidentified function in an alternative form, and be directed to documentation of the supported list of functions and their string identifiers. The output of this step will be a list of primary arithmetic operators, variable identifiers, integer identifiers for functions, and the special character "#" denoting the placement of the next function in the order as they appear in the prompts. The next step of the parser will be to reorder this list obeying standard order of operations, corresponding to the order in which the algorithm will evaluate the functions, and generate a tuple of these identifiers *for each input from the prompt*, meaning that if the user inputs 4 sub-functions, then there will be 4 tuples for each sub-function that are to be considered in the order that they are inputted.

The second component of the parser takes in these tuples and uses them to instantiate corresponding data structures which we will have in the form of a class. The primary class from which other classes inherit from will be the "function class," instantiated with a sole input of the order of operations tuple. This class has two key methods: a method for evaluating the function value using the inputted tuple for instructions, and calling on the relevant mathematical functions in the python math library or `numpy`, and a method for evaluation of the function's derivative, which will be able to evaluate the derivative of the inputted function using the input tuple with respect to *all* variables save those concerning the special character "#," which will be saved for later. There will be an instance of the function class for each sub-function provided by the user. These instances will be handed off to the next stage of the program, which forms the true auto-differentiation routine.

Given that the functions have been aptly described by classes that provide instructions on how to evaluate them and their derivatives, and are ascribed in an order that obeys both order

of operations as well as function hierarchy, we are now prepared to carry out autodifferentiation. At this point, via user prompt, the algorithm decides whether or not to carry out the autodifferentiation task as a backpropogation routine or else a standard autodifferentiation request (see Figure 4). The algorithm defaults to forward mode organization, but can be performed in reverse mode via a rearrangement of the parameters, once again by user prompt. The routine evaluates the function and its derivative at the specified position, then evaluates in the proximity of the point to generate values for the graphing interface, one of the primary outputs. At the end of this loop, a list of floating point or double values comes from the autodifferentiation routine, and is passed to the third and final stage of the algorithm.

The final stage of the algorithm (see Figure 5) will deliver the contents of these floating and double lists to the user. The function and its derivative evaluation will be delivered in their full values in the form of standard output, or in a text box on a graphical interface. The generated points contiguous to the inputted point will be plotted via `matplotlib` for the user to view, along with the values at the input point, plotted in a distinct fashion so that it is conspicuous. This represents the end of one full cycle of the program, and the program will prompt the user if it would like to return to the beginning.

6 FEEDBACK

6.1 Milestone 1

Salient points from the teaching staff feedback are listed in bullet format below. Our responses follow each bullet point, with a link to the fixed portion of the document when applicable.

6.1.1 How to Use

- Our base expectations here are that you provide a python package. You could definitely offer a command line interface, but I think it makes sense to begin development with the assumption that a user will be importing your package within a python file. I think that the Toy AD implementation from HW4 is a good place to start.

  $\Longrightarrow$ We will tailor our approach going forward, but still plan to incorporate CLI at this stage of our development.

- I think there was a lot of confusion about the requirement for some sort of UI which stemmed from the AD visualizer that we used in HW4. Again, there is no expectation for any interface at all. If you wanted to provide one, that could potentially be part of your extra feature, but we should talk more about that once you've completed Milestone 2.

  $\Longrightarrow$ We left this portion in our documentation, but will revise the section after Milestone 2 if we decide whether or not to incorporate a UI.

6.1.2 Software Organization

- I'd like to see maybe a few examples of importing and calling the methods / fake code blocks, but what you have is sufficient.

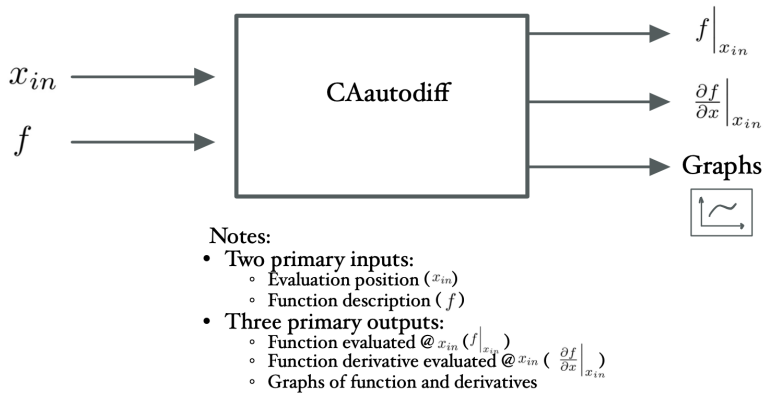
  $\Longrightarrow$ We will incorporate this in our updated Milestone 2 documentation.

- It's probably easier to just provide your package via PyPi / PIP than it is to have a bash install. There's a quick and easy tutorial on doing that as part of the optional PP8 section, but it honestly shouldn't take too long and would be cleaner in my opinion.

  $\Longrightarrow$ Updates made to sections 4.4 and 4.5.

7 FIGURES

Basic Structure (figure 1)



Notes:
- Two primary inputs:
  - Evaluation position (x_{in})
  - Function description (f)
- Three primary outputs:
  - Function evaluated @ x_{in} ($f\big|_{x_{in}}$)
  - Function derivative evaluated @ x_{in} ($\frac{\partial f}{\partial x}\big|_{x_{in}}$)
  - Graphs of function and derivatives

Figure 1

Internal working of the Autodiff (figure 2)

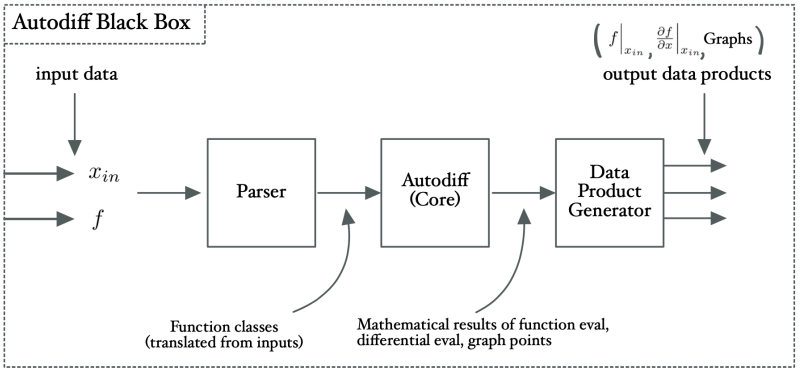
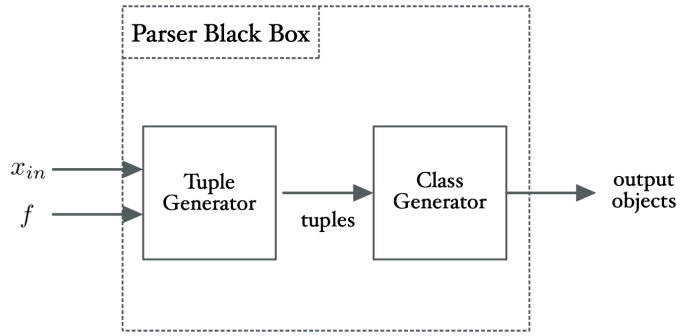


Figure 2

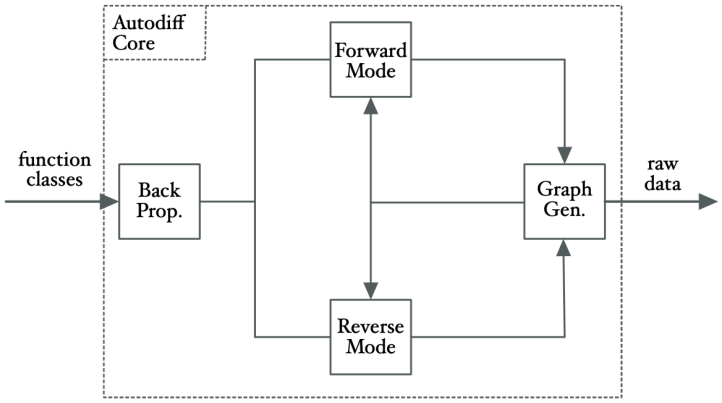
Internal working of Parser
(figure 3)



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 3

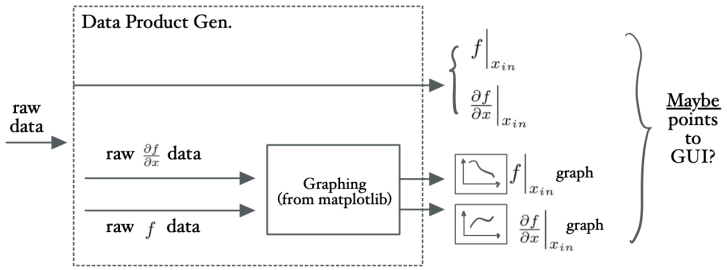
Internal working of Autodiff (core) black box
(figure 4)



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 4

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 5