Assignment 4

CmpE 250, Data Structures and Algorithms, Fall 2017

Instructors: H. L. Akın, A. T. Cemgil
TA: Alper Alimoğlu, Özlem Şimşek
SA: Yusuf Hakan Kalaycı, Abdullatif Köksal

Due: 21 Dec. 2017, 23:59 Sharp

1 Automatic Differentiation

Automatic differentiation is a method for evaluating the rate of change in the numerical output of a program with respect to the rate of change in its input. We start with an example:

$$f(x) = \cos(x)\sin(x)$$

When we want to evaluate the function numerically at a specific x, say $x = 1$, we can implement a computer program like:

$$x = 1$$

$$f = cos(x) * sin(x)$$

or

```
def g(x):
    return cos(x)*sin(x)
x = 1
f = g(x)
```

Now, suppose we need the derivative as well, that is how much f changes when we slightly change x. For this example, it is a simple exercise to calculate the derivative symbolically as:

$$f'(x) = \cos(x)\cos(x) - \sin(x)\sin(x) = \cos(x)^2 - \sin(x)^2$$

and code this explicitely as:

```
x = 1
df_dx =  cos(x)*cos(x) - sin(x)*sin(x)
```

But, could we calculate the derivative without coding it up explicitely, that is without symbolically evaluating it a priori by hand? For example, can we code just:

```
x = 1
f = my.cos(x)*my.sin(x)
df_dx = f.derivative()
```

or

```
def g(x):
    return my.cos(x)*my.sin(x)
x = 1
f = g(x)
df_dx = f.derivative()
```

to get what we want? Perhaps by overloading the appropriate variables, functions and operators? The answer turns out to be yes and it is a quite fascinating subject called **automatic differentiation**. Interestingly, this algorithm, known also as **backpropagation**, is in the core of today's artificial intelligence systems. See https://www.youtube.com/watch?v=aircAruvnKk for an introduction to a particular type of model, known as a neural network.

To symbolically evaluate the derivative, we use the chain rule. The chain rule dictates that for

$$f(x) = g(h(x))$$

, the derivative is given as

$$f'(x) = g'(h(x))h'(x)$$

We could implement this program as:

```
x = 1
h = H(x)
g = G(h)
f = g
```

where we have used capital letters for the functions – beware that the function and its output is always denoted with the same letter in mathematical notation. To highlight the underlying mechanism of automatic differentiation, we will always assign the output of a function to a variable so we will only think of the rate of change of a variable with respect to another variable, rather than 'derivatives of functions'. To be entirely formal we write:

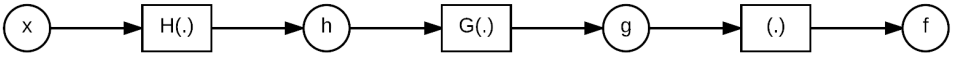

Figure 1:

```
x = 1
h = H(x)
g = G(h)
f = identitiy(g)
```

and denote the identity function as $(\cdot)$. This program can be represented also by the directed computation graph denoted in Figure 1:

The derivative is denoted by:

$$f'(x) = \frac{df}{dx}$$

As we will later use multiple variables, we will already introduce the partial derivative notation, that is equivalent to the derivative for scalar functions:

$$f'(x) = \frac{\partial f}{\partial x}$$

The chain rule, using the partial derivative notation can be stated as:

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{\partial h}{\partial x}\frac{\partial g}{\partial h}\frac{\partial f}{\partial g} && (1)\\ &= h'(x)g'(h(x)) \cdot 1 && (2) \end{aligned}$$

This quantity is actually just a product of numbers, so we could have evaluated this derivative in the following order:

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{\partial h}{\partial x}\left(\frac{\partial g}{\partial h}\left(\frac{\partial f}{\partial g}\frac{\partial f}{\partial f}\right)\right) && (3)\\ &= \frac{\partial h}{\partial x}\left(\frac{\partial g}{\partial h}\frac{\partial f}{\partial g}\right) && (4)\\ &= \frac{\partial h}{\partial x}\frac{\partial f}{\partial h} && (5)\\ &= \frac{\partial f}{\partial x} && (6) \end{aligned}$$

where we have included $\partial f/\partial f = 1$ as the boundary case.

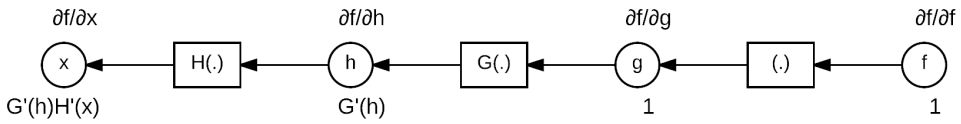

Figure 2:

So, we could imagine calculating the derivative using the following program:

```
df_df = 1
df_dg = 1 * df_df
df_dh = dG(h) * df_dg
df_dx = dH(x) * df_dh
```

If g and h are elementary functions, their derivatives are known in closed form and can be calculated from their input(s) only.

As an example, consider:

$$f(x) = \sin(\cos(x))$$

The derivative is:

$$\frac{\partial f}{\partial x} = -\sin(x)\cos(\cos(x))$$

```
df_df =  1
df_dg =  1 * df_df
df_dh =  cos(h) * df_dg
df_dx = -sin(x) * df_dh
```

As $h = \cos(x)$, it can be easily verified that the derivative is calculated correctly.

1.1 Functions of two or more variables

When we have functions of two or more variables the notion of a derivative changes. For example, when:

$$g(x_1, x_2)$$

we define the partial derivatives:

$$\frac{\partial g}{\partial x_1} \quad , \quad \frac{\partial g}{\partial x_2} \tag{7}$$

The collection of partial derivatives can be organized as a vector. This object is known as the gradient and is denoted as:

$$\nabla g(x) \equiv \begin{pmatrix} \frac{\partial g}{\partial x_1} \\ \frac{\partial g}{\partial x_2} \end{pmatrix} \tag{8}$$

When taking the partial derivative, we assume that all the variables are constant, apart from the one that we are taking the derivative with respect to.

For example,

$$g(x_1, x_2) = \cos(x_1)e^{3x_2}$$

When taking the (partial) derivative with respect to x_1, we assume that the second factor is a constant:

$$\frac{\partial g}{\partial x_1} = -\sin(x_1)e^{3x_2}$$

Similarly, when taking the partial derivative with respect to x_2, we assume that the first factor is a constant:

$$\frac{\partial g}{\partial x_2} = 3\cos(x_1)e^{3x_2}$$

The chain rule for multiple variables is in a way similar to the chain rule for single variable functions but with a caveat: the derivatives over all paths between the two variables need to be added.

Another example is:

$$f(x) = g(h_1(x), h_2(x))$$

Here, the partial derivative is:

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial h_1}\frac{\partial h_1}{\partial x} + \frac{\partial g}{\partial h_2}\frac{\partial h_2}{\partial x}$$

The chain rule has a simple form:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

To see a concrete example of a function of form $f(x) = g(h_1(x), h_2(x))$, consider:

$$f(x) = \sin(x)\cos(x)$$

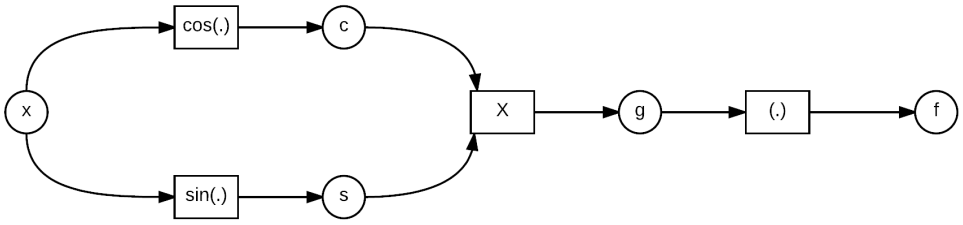

Figure 3:

We define:

$$h_1(x) = c = \cos(x) \tag{9}$$
$$h_2(x) = s = \sin(x) \tag{10}$$
$$g(c, s) = g = c \times s \tag{11}$$
$$f = g(c, s) \tag{12}$$

that is equivalent to the following program, written deliberately as a sequence of scalar function evaluations and binary operators only:

```
x = 1
c = cos(x)
s = sin(x)
g = c * s
f = g
```

This program can be represented by the directed computation graph given in Figure 3:

The function can be evaluated by traversing the variable nodes of the directed graph from the inputs to the outputs in the topological order. At each variable node, we merely evaluate the incoming function. Topological order guarantees that the inputs for the function are already calculated.

It is not obvious, but the derivatives can also be calculated easily. By the chain rule, we have:

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{\partial f}{\partial g}\frac{\partial g}{\partial c}\frac{\partial c}{\partial x} + \frac{\partial f}{\partial g}\frac{\partial g}{\partial s}\frac{\partial s}{\partial x} && (13)\\ &= 1 \cdot s \cdot \sin(x) + 1 \cdot c \cdot (-\cos(x)) && (14)\\ &= 1 \cdot \sin(x) \cdot \sin(x) + 1 \cdot \cos(x) \cdot (-\cos(x)) && (15)\\ & && (16) \end{aligned}$$

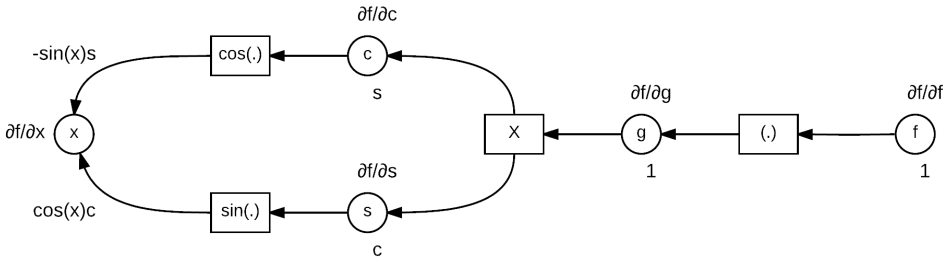

Figure 4:

The derivative could have been calculated numerically by the following program:

```
df_dx = 0, df_ds = 0, df_dc = 0, df_dg = 0
df_df = 1

df_dg +=  df_df              // df/dg = 1
df_dc +=  s * df_dg          // dg/dc = s
df_ds +=  c * df_dg          // dg/ds = c
df_dx +=  cos(x) * df_ds     // ds/dx = cos(x)
df_dx += -sin(x) * df_dc     // dc/dx = -sin(x)
```

Note that the total derivative consists of sums of several terms. Each term is the product of the derivatives along the path leading from f to x. In the above example, there are only two paths:

1. f, g, c, x
2. f, g, s, x

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial c}\frac{\partial c}{\partial x} + \frac{\partial f}{\partial g}\frac{\partial g}{\partial s}\frac{\partial s}{\partial x}$$

It is not obvious in this simple example but the fact that we are propagating backwards makes us save computation by storing the intermediate variables.

This program can be represented by the directed computation graph given in the Figure 4:

Note that during the backward pass, if we traverse variable nodes in the reverse topological order, we only need the derivatives already computed in

previous steps and values of variables that are connected to the function node that are computed during the forward pass. As an example, consider:

$$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial c}$$

The first term is already available during the backward pass. The second term needs to be programmed by calculating the partial derivative of $g(s, c) = sc$ with respect to c. It has a simple form, namely s. More importantly, the numerical value is also immediately available, as it is calculated during the forward pass. For each function type, this calculation will be different but is nevertheless straightforward for all basic functions, including the binary arithmetic operators $+, -, \times$ and $\div$.

2 Statement

In this project, you will write a program that gets four input file names. The first file will be the function definition file, the second will be the input values for which the function and its derivative is to be calculated.

You will calculate function and partial derivatives for given input values and print the results to two output files one for value of function (third input) and one for partial derivatives (fourth input).

3 Input/Output Format

3.1 Function Definition File

The function definition file will have the following format:

```
<input>
<input>
...
<input>
<output>
<assignment>
<assignment>
...
<assignment>
```

Context free grammar of the $< input >$, $< output >$ and $< assignment >$ will be following:

<input> -> input <var>
<output> -> output <var>
<assignment> -> <var> = <function> <var> <var>
<assignment> -> <var> = <function> <var>
<assignment> -> <var> = <var>

To make it clear, let's describe the format verbally.

- Each input is described with two words where the first word is "input" and the second word is the name of the input variable.
- The only output is described with two words where the first word is "output" and the second word is the name of the output variable.
- Each assignment can be one of the following format:
  - $< var > = < function > < var > < var >$ where $< var >$'s are variable names and $< function >$ is function name. This format is designed for functions with two parameters.
  - $< var > = < function > < var >$ where $< var >$'s are variable names and $< function >$ is function name. This format is designed for functions with one parameters.
  - $< var > = < var >$ where $< var >$'s are variable names. This format is designed for assignments.

Functions with single parameters can be one of the followings:
cos
sin
tan
acos
asin
atan
exp
log
sqrt

Functions with two parameters can be one of the followings:
add
mult
subs
divide
pow

where add, mult, subs and divide are basic binary operations.

3.2 Input Values File

The input values file will have the following format:

```
<input 1> <input 2> ... <input n>
<value 1> <value 2> ... <value n>
<value 1> <value 2> ... <value n>
<value 1> <value 2> ... <value n>
...
```

$< input1 >, < input2 > ... < inputn >$'s will be the names of input parameters and they are separated by single space character.

$< value1 >, < value2 > ... < valuen >$'s will be the values of the corresponding parameter and these values are separated by single space character. For each line you will calculate output of described function and partial derivatives with respect to the each input variable.

3.3 Output Values File

Your program will have two output files: Function values and the derivatives (up to 5 digits of precision). First output file is the output values file and the output values file will have the following format:

```
<output>
<value 1>
<value 2>
<value 3>
...
```

$< output >$ is the name of output variable. $< valuei >$ is the value of function with the corresponding input values given in the input values file.

3.4 Derivative Values File

Your second output file will be the file of derivative values for corresponding input values. The derivative values file will have the following format:

```
<deriv 1> <deriv 2> ... <deriv n>
<value 1> <value 2> ... <value n>
<value 1> <value 2> ... <value n>
<value 1> <value 2> ... <value n>
...
```

$< deriv1 >, < deriv2 > ... < derivn >$'s are the names of derivatives (for more details please look the example). $< value1 >< value2 > ... < valuen >$'s are the values of derivatives with respected to corresponding variable and with the corresponding input values given in the input values file.

4 Example

An example is the following for the function:

$$f(x_1, x_2) = \sin(2x_1)\cos(x_1x_2)$$

Corresponding function definition file will look like:

```
input x_1
input x_2
output f
t_0 = mult 2 x_1
t_1 = sin  t_0
t_2 = mult x_1 x_2
t_3 = cos  t_2
t_4 = mult t_1 t_3
f   = t_4
```

If we want to evaluate the function on values $(x_1, x_2) = \{(0, 0), (1.2, -3), (5, 5)\}$, the example file format will be:

```
x_1 x_2
0 0
1.2 -3
5 5
```

Your output values file must be similar to following results where the values must be near to corresponding original ones up to required precision constant:

```
f
0.0
-0.60573
-0.53924
```

The partial derivatives are:

$$\frac{\partial f}{\partial x_1} = 2\cos(2x_1)\cos(x_1x_2) - \sin(2x_1)\sin(x_1x_2)x_2$$

$$\frac{\partial f}{\partial x_2} = -\sin(2x_1)\sin(x_1x_2)x_1$$

Hence, the output file will be:

```
df/dx_1  df/dx_2
2.0  -0.0
0.24682  -0.35869
0.20232  -0.36001
```

5 Grading

Grading of this project is based on two cases for any test case.

1. If the printed output values in the output values file are near to corresponding original ones up to required precision constant(10^{-5}), then you will get 35% points of that test case.

2. If the printed derivative values in the derivative values file are near to corresponding original ones up to required precision constant(10^{-5}), then you will get 65% points of that test case.

Your score will be the sum of collected points from each test case and each test case contribute the project score equally. Maximum score can be up to 100.

6 Implementation Details

1. Your program will be compiled with **cmake CMakeLists.txt && make** command. Therefore, if you add new files, you have to check CMakeLists.txt is updated accordingly so that your code auto-compiles. Note that it is also fine to code in a single file in this project (Because I am assuming you already know what .h and .cpp files are and how to use those)

2. I will execute your program with **./project4 functionDefinitionFile inputValuesFile outputValuesFile derivativeValuesFile** command. So, use command line arguments in your main function accordingly.

3. In order to make the implementation easier a boilerplate code is prepared and it can be found in the Github repository. You may use it to implement this project.

4. Either you use the boilerplate code or not, designing and implementing project in a object oriented fashion is strongly recommended. This will make your implementation clear and easy to develop.

7 Project Guidelines

1. Warning: All source codes are checked automatically for similarity with other submissions and also the submissions from previous years. Make sure you write and submit your own code.

2. Your program will be graded based the correctness of your output and the clarity of the source code. Correctness of your output will be tested automatically so make sure you stick with the format described above.

3. There are several issues that makes a code piece 'quality'. In our case, you are expected to use C++ as powerful, steady and flexible as possible. Use mechanisms that affects these issues positively.

4. Make sure you document your code with necessary inline comments, and use meaningful variable names. Do not over-comment, or make your variable names unnecessarily long.

5. Try to write as efficient (both in terms of space and time) as possible. Informally speaking, try to make sure that your program completes in meaningful amount of time.