Project Proposal: geocold Ray Tracer (Differentiable?) in C++

Authors: Prakash Chaulagain, Nishar Arjyal, and Pramish Paudel

Roll Numbers: 076BCT045, 076BCT042, 076BCT047

Submitted to the Department of Electronics and Computer Engineering in Partial Fulfillment of the Requirements of the 3rd Year **Computer Graphics** Course at
Pulchowk Campus
IOE, Tribhuwan University

June 23, 2022

Accepted by: ....

Mr. Basanta Joshi
Assistant Professor at
The Department of Electronics and Computer Engineering

Date of Submission: June 23, 2022
Expected Date of Completion: August, 2022

Mathematical Toolkit

by Prakash Chaulagain, Nishar Arjyal, and Pramish Paudel
Submitted to the Department of Electronics and Computer Engineering
on June 23, 2022
in Partial Fulfillment of the Requirements for the 3rd Year Computer Graphics Course in Computer Engineering

Abstract

Derivatives and matrices seem to be ubiquitous in all of mathematical optimization and machine learning nowadays. In such times, it becomes all the more important that the average computer programmer has access to a software package that is powerful yet simple to learn in order to do fast gradient computations alongside a linear algebra package that- while-using feels native to the programming language of choice. The purpose of this project is to come up with one such mathematical package to fulfil the requirements of academics for either soft-core academic work or small-scale projects that require some form of mathematical optimization in C++. With MathematicalToolkit we aim to give the user a native C++ experience in calculating derivatives of their functions.
Several different methods exist for calculating the derivatives of functions. Numerical and Symbolic are the first that come to mind. However, when it comes to computer programming, the programming world seems to have unanimously decided that the best method for a computer to differentiate functions is by using a set of techniques called Automatic Differentiation (see [5]). Automatic Differentiation (AD) is simply the most efficient family of techniques for accurately calculating the derivatives of functions. Several techniques exist in AD (see [2]), however, the goal of this project is to implement only the Forward-Mode Automatic Differentiation using the operator overloading approach. This project also aims to build a wrapper linear algebra sub-package for simple yet efficient matrix and vector related computations.

Table Of Contents

1 Acknowledgement

Our project idea is the product of excellent supervision of all of our instructors, most notably our lecturer Mr. Basanta Joshi. We could not have been able to develop interest in computer graphics as a field of study without the constant inspiration provided to us by all of our lecturers and lab instructors and assistants. Some of the credit should also go to the college administration for their efforts in the smooth functioning of all of our classes and labs safely and securely despite the unprecedented times of the pandemic.

2 Objectives

- To bring Automatic Differentiation in a mathematical library that is suitable to use for academics.
- To analyze Object Oriented Programming design.
- To get familiar with program optimization and safe memory-handling practices.

3 Introduction

The most popularly used method for the computation of derivatives of functions or mathematical expressions in computer program form when it comes to mathematical optimization problems or machine learning is *automatic differentiation*, also called *algorithmic differentiation* which is the major subject matter of this project.

Conventionally, most of the algorithms of optimization have relied heavily on computing derivatives and gradients of functions (see [8][1]). Multiple implementations of automatic differentiation exist in various programming languages such as in C++ (Carpenter et al., 2015 [3]) as well as some higher level programming languages like Julia (see [9]) However, most of these implementations are built to be used specifically in machine learning and not for academic work. Furthermore, most such implementations are the reverse mode automatic differentiation which is more suitable in machine learning but perhaps less suitable in mathematics as we normally use. We aim to provide a very naive and incredibly hackable AD package that is usable for the clueless as much as it is for the pros.

3.1 Scalar Dual Numbers and Forward Mode AD

The *scalar dual number* type implemented in our library is defined as:

$$\boxed{f(x + y\epsilon) = f(x) + yf'(x)\epsilon + \mathcal{O}(\epsilon^2)} \tag{1}$$

The definition of the scalar dual number in equation 1 contains a primal/value part 'x' and a derivative part 'y'. Following eqn. 1, we obtain the derivative

[1] https://mitpress.mit.edu/books/algorithms-optimization

of any function through eqn. 2

$$\boxed{f'(x) = \frac{f(x + y\epsilon) - f(x)}{y\epsilon}} \tag{2}$$

By defining some dual number arithmetic through operator overloading such as:

$$\boxed{(x_1 + y_1\epsilon) + (x_2 + y_2\epsilon) = (x_1 + x_2) + (y_1 + y_2)\epsilon} \tag{3}$$

We can achieve this easily by overloading the '+' operator over our custom scalar '*Dual*' type.

The eqn.4 is the definition of the product of two dual types.

$$\boxed{(x_1 + y_1\epsilon) \times (x_2 + y_2\epsilon) = (x_1x_2) + (x_1y_2 + x_2y_1)\epsilon} \tag{4}$$

So, basically we are defining the basic derivative formulae on the dual type, and for chained functions, we define a set of chain rules. In this way, we can obtain the derivative of all scalar functions that depend on only one variable. The ϵ in all of these formulae is called the $machine - epsilon$ defined as: $\epsilon^2 = 0$ in the computer. The epsilon is defined as such: $1.0 + \epsilon == 1.0$ returns a true in the computer program. So, basically, when we add or subtract a number from or to an ϵ we have done literally no computation. Further, all higher powers of ϵ are zero.

3.2 Multidimensional Dual Numbers and Vector Forward Mode AD

The scalar type mentioned above works suitably for scalar one-variable functions; however, better approach can be taken in the case of multivariate functions.
Let's analyze how we can differentiate a function of two variables using our scalar dual numbers. The L_2 norm of a vector $\overrightarrow{x}$ of length n is defined to be (the indices are chosen to be the same as those used in programming normally).

$$\boxed{||\overrightarrow{x}|| = \left(\sum_{i=0}^{i=n-1} x_i^2\right)^{\frac{1}{2}}} \tag{5}$$

Let us try finding the gradient of the square of this functions when the input has two elements.

$$\boxed{f(x_0, x_1) = x_0^2 + x_1^2} \tag{6}$$

First, the partial derivative with respect to x_0 is found by noticing that the derivative of x_0 w.r.t. x_0 itself is 1 and the derivative of x_1 w.r.t. x_0 is 0.
So basically, we are passing the dual number into the function which returns the value alongside the derivative as a dual number.

$$f(x_0 + \epsilon, x_1) = (x_0 + \epsilon)^2 + x_1^2 \tag{7}$$

$$f(x_0 + \epsilon, x_1) = x_0^2 + 2x_0\epsilon + \epsilon^2 + x_1^2 \tag{8}$$

$$f(x_0 + \epsilon, x_1) = (x_0^2 + x_1^2) + 2x_0\epsilon \tag{9}$$

Notice from the second equation above to the third, we used the definition that ϵ^2 is 0 and so are all higher powers of ϵ
So, the value of $f(x_0, x_1)$ at (x_0, x_1) is $x_0^2 + x_1^2$ while the partial derivative of $f(x_0, x_1)$ w.r.t. x_0 is $2x_0$ This method becomes clearly very rigorous if the input has a large size when it comes to making gradient computations. We only had to calculate the derivative w.r.t. one input here. For the gradient, we would need to set the derivative part of x_0 equal to zero just like we set the derivative part of x_1 to zero in the above equations and set the derivative part of x_1 to 1. In both the cases, we would get a dual number with the same primal/value parts but the derivative parts would separately be the partial derivative of the function with respect to x_0 and x_1. So, we can reduce redundancy by wrapping our input and the dual number into a vector. We introduce the multidimensioanl dual number type in our package. The multidimensional dual number type implemented in MathematicalToolkit is very much so based on the paper [9] and is defined as such for a scalar function:

$$f(x + \sum_{i=1}^{k} y_i\epsilon_i) = f(x) + f'(x)\sum_{i=1}^{k} y_i\epsilon_i \tag{10}$$

The product of all $\epsilon_i\epsilon_j$ is zero by definition. For the case in which the function depends upon multiple variables:

$$\overrightarrow{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_k \end{bmatrix} \to \overrightarrow{x_\epsilon} = \begin{bmatrix} x_1 + \epsilon_1 + 0\sum_{n=2}^{k} \epsilon_n \\ \vdots \\ x_i + \epsilon_i + 0\sum_{n\neq i} \epsilon_n \\ \vdots \\ x_k + \epsilon_k + 0\sum_{n=1}^{k-1} \epsilon_n \end{bmatrix} = \begin{bmatrix} x_1 + \epsilon_1 \\ \vdots \\ x_i + \epsilon_i \\ \vdots \\ x_k + \epsilon_k \end{bmatrix} \to f(\overrightarrow{x}_\epsilon) = f(\overrightarrow{x}) + \sum_{i=1}^{k} \frac{\partial f(\overrightarrow{x})}{\partial x_i}\epsilon_i \tag{11}$$

Now, we can calculate the gradient of $f(x_0, x_1)$ from before as such: First, for the first component of the gradient:

$$f\left(\begin{bmatrix} x_0 + \epsilon_0 \\ x_1 + \epsilon_1 \end{bmatrix}\right) = x_0^2 + 2x_0\epsilon_0 + x_1^2 + 2x_0\epsilon_1 = (x_0^2 + x_1^2) + \epsilon_0(2x_0) + \epsilon_1(2x_1)$$

So, in one single pass, we have calculated the value of $\nabla f(\overrightarrow{x})$ to be : $\langle 2x_0, 2x_1 \rangle$.

4 Existing Systems

Several implementations of the Forward and Vector Forward Mode AD or even Reverse Mode AD exist already. Perhaps the most popular such library is TensorFlow(see [1])[2] (some people may not have been acquaintained with the fact that TensorFlow is an AD software but it is). TensorFlow applies a *trace-based* implementation of the reverse-mode AD. A popular such tool implemented using operator overloading in C++ is ADOL-C (see[10])[3]. A popular high-level implementaton of the vector forward-mode AD is in a Julia package called ForwardDiff.jl (see [9]) [4]. The Stan Math Library (see [3]) is a C++ implementation of reverse mode automatic differentiation ([5]). Fast AD (see[11]) is a C++ AD library based on Expresion Templates.

[2] https://github.com/tensorflow/tensorflow
[3] https://github.com/coin-or/ADOL-C
[4] https://github.com/JuliaDiff/ForwardDiff.jl

5 Proposed System

5.1 Description

The user should be able to use our package after downloading and then doing a '*build*' of the package on their system. The library should be *callable* in one's program by doing something as simple as:
#**include** <MathematicalToolkit>

The user should be able to write the following code and have their gradients computed:

Listing 1: What Using MathematicalToolkit Would Feel Like

```
#include <iostream>
#include <MathematicalToolkit> //including the library header file
grad::NDual f(grad::NDual x,grad::NDual y)
{
   return x*x + y*y;    //f(x,y) = sq(x) + sq(y)
}
int main()
{
   double eps1[] = {1,0}
   double eps2[] = {0,1};
   grad::NDual x(1.0,eps1);
   grad::NDual y(1.0,eps2); //will give the gradient of x*x+y*y at
   f(x,y).grad::gradient(); //x = 1.0, y = 1.0
}
```

A program as simple as this should give to the user the gradient of the function. Obviously, the final thing that we intend to serve would be capable to do more than just this; however, this is only a simple glimpse of what our library is supposed to bring. We intend our library to work comfortably with numerous most common mathematical functions and data types.
The library is supposed to be type generic as much as possible and one of the goals is to write the source code of the library in a way that is possible and easy for everyone to read and get a grasp of what is going on under the hood. Our package is also supposed to provide to the user a simple linear algebra wrapper library that makes it possible to solve systems of linear equations, as well as constructing Vandermonde matrices which find their usage in *Polynomial Interpolation*, among many other features one would

normally expect of a linear algebra library.
Our program should be made available at GitHub as an Open Source Project. The directory structure followed by our project should be relatively simple and follow the common directory tree structure of most of the C++ libraries.

The package is suppposed to have the following directory structure:

```
MathematicalToolkit
├── docs
├── include
├── src
└── test
```

Thus, following the industry standard of creating C++ libraries.
In the final product, we intend to show examples of what are the possibilities that one can unlock using our library. We intend to show some visualization of *Polynomial Curve Fitting* as well as some *Optimization Algorithms* at work using MathematicalToolkit.

5.2 System Block Diagram

The following is a block diagram of our system:

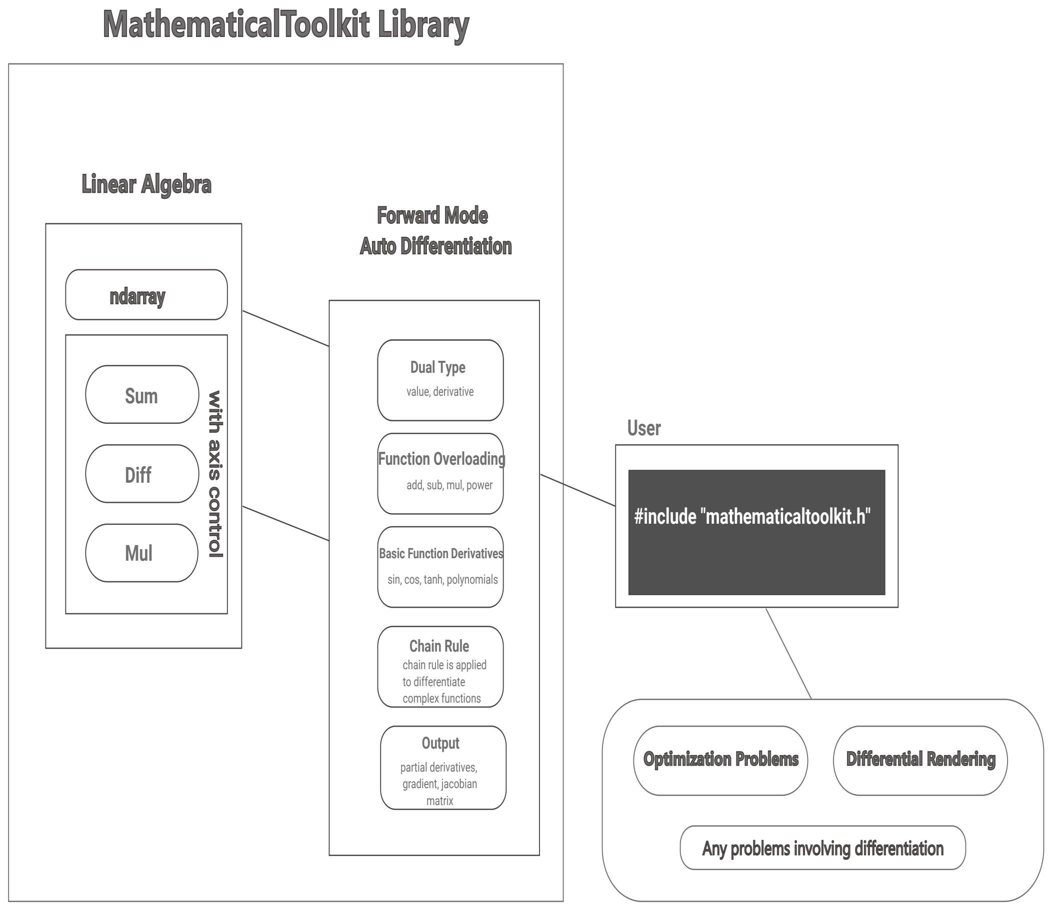

Figure 1: Block Diagram for our System

6 Methodology

The Graph in Figure 2 shows the computation of the gradient of the 3-variable square of the L_2 norm function in a single pass. By creating a Multidimensional Dual class which has a primal value alongside a gradient vector, and then overloading the different arithmetic operators that we normally come across in everyday computations over this class, we can create a fully functioning *gradient-computing* software. We, could use the existing standard vector (std::vector) implementation of the C++ standard library,

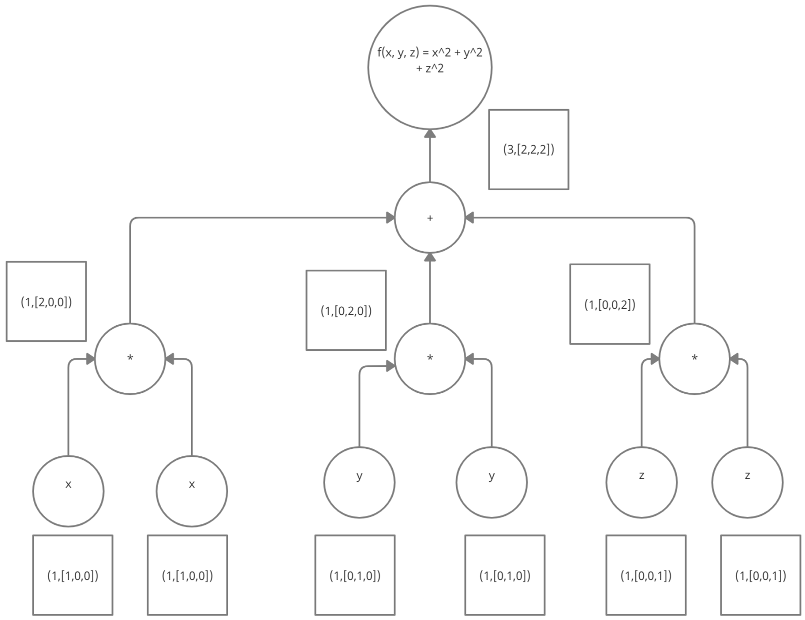

Figure 2: Forward Pass of the Primal and Gradient Values Over the Computational Graph

however, since our project for this semester is a very ***Proof-Of-Concepts***, we inted to try out our own vector class for this library.

Listing 2: How we could create our Dual Number class:

```
template <size_t N, typename T>
class NDual
{
    private:
        T primal;
        std::vector<T> grad(N);
    public:
        //relevant constructors, destructors, ...
        //overloading the operators over the dual type
};
```

The simple class template in the above listing can form the basis of our entire library. We intend to learn and analyze object oriented best practices to come up with a memory safe and fairly optimized library.

7 Project Scope

Differentiation of functions, expressions written as computer programs is extremely important. We have had ways of calclating the square root of something in computers forever. In C++, one could use the *sqrt*() function of the standard library. However, when it comes to differentiations, no such features are available. It becomes incredibly cumbersome for a programmer or any computer user to write their own code to differentiate a function and that's the issue we intend to solve with MathematicalToolkit.

As data sets grow larger and larger, the fields like deep learning(see[4]) require sophisticated Mathematical software for their usage. Though, we cannot promise our project in the Second Year of Engineering to match advanced implementations like Flux (see[6],[7]) and TensorFlow (see[1]), we hope to get a grasp of most of these ideas under the object oriented paradigm which also happens to be the industry standard.

8 Project Schedule

The project is under development currently [5]. We intend to share our final project with the class come presentation day and it is supposed to be finished by the month of August. The project should take about three and a half weeks for its completion. As we work on our library, we intend to also create some examples that can be helpful for users to see the capabilities of our library.

[5]https://github.com/yamisukehiro27/MathematicalToolkit

References

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: `1603.04467 [cs.DC]`.

[2] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: `1502.05767 [cs.SC]`.

[3] Bob Carpenter et al. *The Stan Math Library: Reverse-Mode Automatic Differentiation in C++*. 2015. arXiv: `1509.07164 [cs.MS]`.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[5] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, 2008. ISBN: 9780898716597.

[6] Michael Innes et al. "Fashionable Modelling with Flux". In: *CoRR* abs/1811.01457 (2018). arXiv: `1811.01457`. URL: `https://arxiv.org/abs/1811.01457`.

[7] Mike Innes. "Flux: Elegant Machine Learning with Julia". In: *Journal of Open Source Software* (2018). DOI: `10.21105/joss.00602`.

[8] Kochenderfer and Wheeler. *Algorithms For Optimization*. 2019. ISBN: 9780262364775.

[9] J. Revels, M. Lubin, and T. Papamarkou. "Forward-Mode Automatic Differentiation in Julia". In: *arXiv:1607.07892 [cs.MS]* (2016). URL: `https://arxiv.org/abs/1607.07892`.

[10] Andrea Walther. "Getting started with ADOL-C". In: *Combinatorial Scientific Computing* (Jan. 2009). DOI: `10.1201/b11644-8`.

[11] James Yang. *FastAD: Expression Template-Based C++ Library for Fast and Memory-Efficient Automatic Differentiation*. 2021. arXiv: `2102.03681 [cs.MS]`.