



# Utilizing Clad derivatives in RooFit

Garima Singh | Jan - June 2022

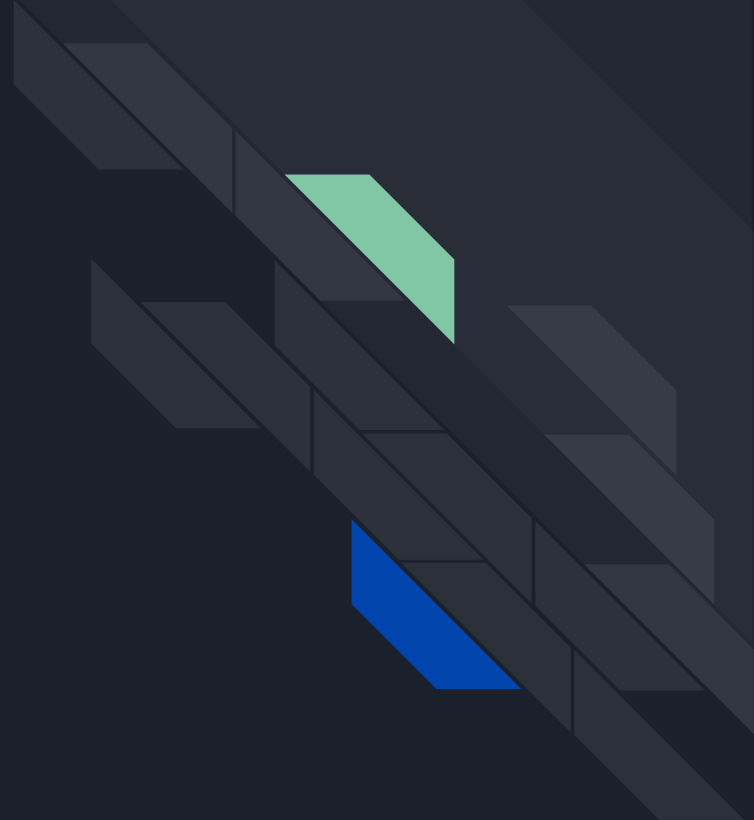
# Today's Agenda

Overview

Major preliminary roadblocks

Possible approaches

Expected timeline of work





# Overview

Roofit is a library that comes packaged in ROOT, it enables the modelling of event data distributions and perform further mathematical analysis on it such as likelihood fits, generating plots etcetera. For all its differentiability needs, Roofit relies solely on numerical differentiation capabilities of the MathCore library in ROOT. Not only is numerical differentiation less accurate and more sensitive to errors, it also does not scale well with larger number of parameters.

There was some work done to parallelize the numerical differentiation calculation and it yielded a significant amount of speedup, but still the other two shortcomings were not mitigated by this effort.<sup>[1]</sup>

This is where automatic differentiation comes in.

[1]: [https://www.epj-conferences.org/articles/epjconf/abs/2020/21/epjconf\\_chep2020\\_06027/epjconf\\_chep2020\\_06027.html](https://www.epj-conferences.org/articles/epjconf/abs/2020/21/epjconf_chep2020_06027/epjconf_chep2020_06027.html)



## Overview Cont.

The main idea of this project is to utilize Clad's automatic differentiation (AD) abilities to generate derivatives for RooFit. AD based derivatives are more accurate, less sensitive to numerical errors and also scale well.



# Preliminary RoadBlocks

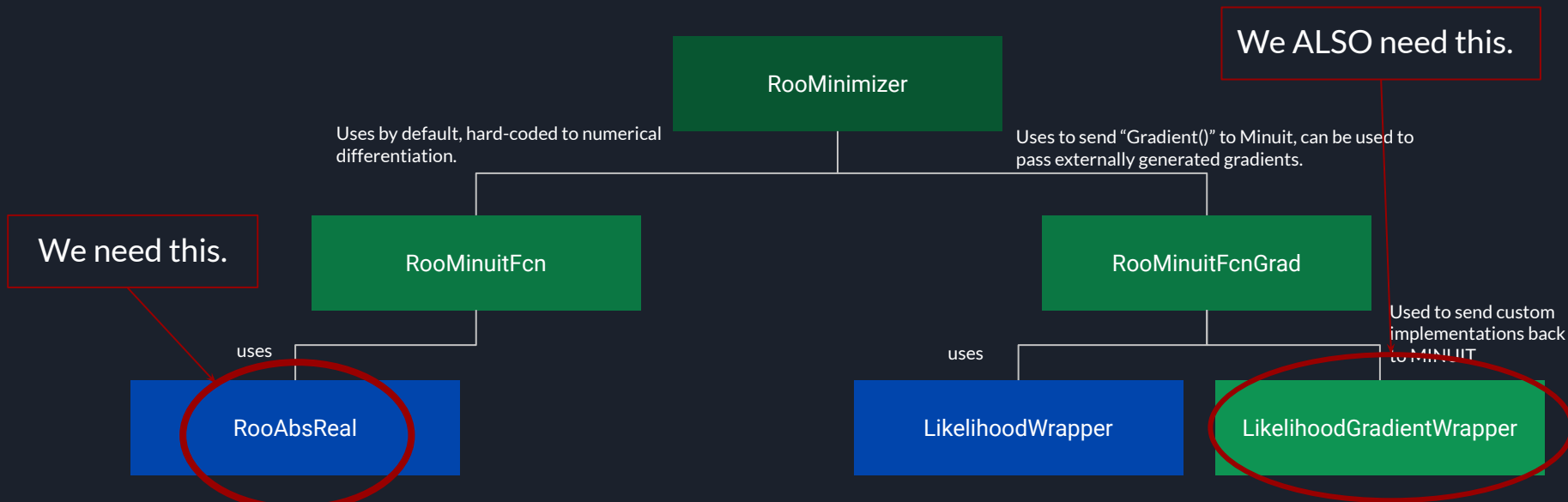
- 01 RooFit is highly optimized for the numerical differentiation use-case. It contains a fair amount of wrappers and syntactic sugar (necessary for caching results for numerical differentiation) over the base elements of the analysis, making it difficult to expose these values to Clad.
- 02 RooFit represents all its calculations in the form of objects and compute graphs. For a fair amount of such objects, extracting the representative C++ code is not straightforward.
- 03 How to bridge the Clad runtime and RooFit runtime.

# Possible Approaches

## Interface design

The first way we approached this problem was to see how the different components connect to the RooFit::RooMinimizer (the place we want to add the clad generated derivatives to).

How do we get components from both the code paths?



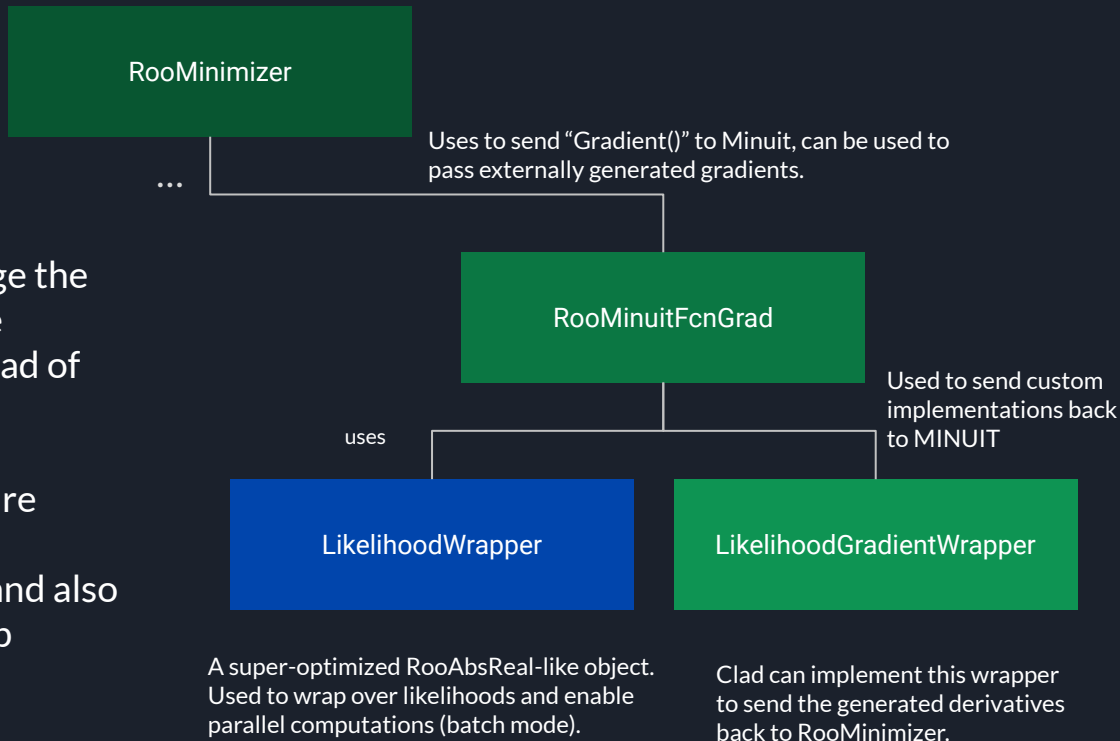
We implement in each **RooAbsReal** an "ExecuteGradient" Method that returns its Gradient or alternatively some representative C++ code (like evaluate()).

A super-optimized **RooAbsReal**-like object. Used to wrap over likelihoods and enable parallel computations (batch mode).

Clad can implement this wrapper to send the generated derivatives back to **RooMinimizer**.

# Possible Approaches Contd.

## Interface design



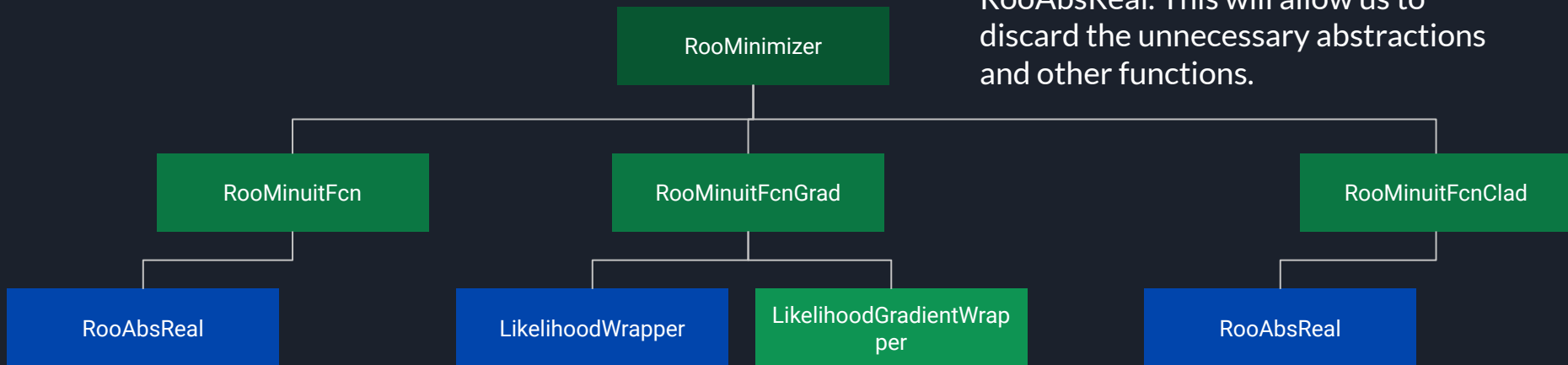
The first idea was to change the second codepath to utilize RooAbsReal Objects instead of RooAbsL.

However, this would require reworking the whole RooMinuitFcnGrad class and also the LikelihoodGradientJob interface.

# Possible Approaches Contd.

## *Interface design*

Another possible approach is to duplicate the RooMinuitFcnGrad into another class that works with RooAbsReal. This will allow us to discard the unnecessary abstractions and other functions.







## Possible Approaches Contd.

### *Clad design*

For any RooFit code to work with clad, we need to generate the C++ code it represents. For single objects this is straightforward, however to compress/squash compute graphs into C++ code becomes slightly complex. After some discussion, we brought the possible approaches down to 2.

1. Hand write the equivalent C++ code for some commonly appearing subgraph/clusters in analyses manually.
2. Let Clad squash the graphs by using some predetermined rule.
3. \* Use chain rule to calculate the derivatives, given we know the derivative of each node individually.



## Possible Approaches Contd.

*Clad design*

### Approach 1

Squash frequently used subgraphs/clusters into one `Roo{xyz}` class that implements a “translate” function. Support calculating only the derivatives of `RooAbsReal` objects that implement the “translate function”.

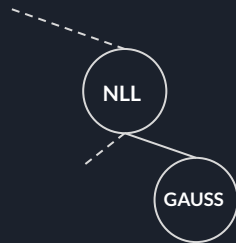
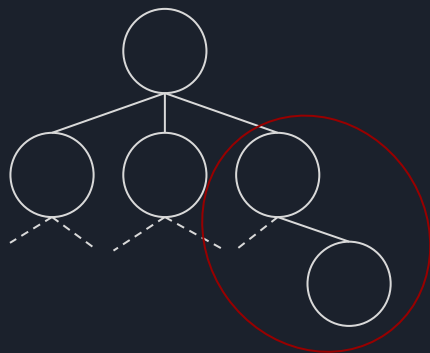
### Approach 2

Use clad to squash the graph by unfolding nested function calls.

# Possible Approaches Contd.

*Clad design : Approach 1 - Squash frequent clusters manually*

While RooFit is a huge codebase with support for a lot of mathematical formulas, it is observed that in most of the analysis, users end up using only a small set of recurrent RooFit objects. It may hence be beneficial to squash these frequently appearing classes manually and then simply implement a function that translates them to C++ code.



```
RooGaussian::Translate()  
+  
RooNLL::Translate()  
=  
RooAutodiffGaussianNLL::Translate()
```

Identify frequently occurring clusters

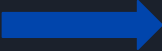
Manually combine the codes together.

# Possible Approaches Contd.

*Clad design : Approach 1 - Squash frequent clusters manually*

```
Double_t RooGaussian::Translate() const
{
    const double arg = x - mean;
    const double sig = sigma;
    return std::exp(-0.5*arg*arg/(sig*sig));
}
```

```
Double_t RooNLLVar::Translate() const
{
    double nll = 0.0;
    for (auto x : *_data)
        nll += -std::log(x);
    return nll;
}
```



```
double RooAutodiffGaussianNLL::Translate() const
override {
    double nll = 0.0;
    for (auto x : *_data) {
        double proba = gaus(x, _mean, _sigma);
        nll += -std::log(proba);
    }
    return nll;
}
```

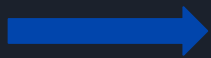
```
clad::gradient(&RooAutodiffGaussianNLL::Translate);
```

# Possible Approaches Contd.

*Clad design : Approach 2 - Squash graphs using Clad*

With how the RooFit computation graphs are computed, we can unfold nested function calls with the following rule

`f(g(x));`



```
f(x) {  
    double result;  
    { // unfold the body of g(x) here  
        // For each return,  
        // replace it with an assign expression  
        // return x + y;  
        result = x + y;  
    }  
    x = result;  
}
```

# Possible Approaches Contd.

## *Clad design : Approach 2 - Squash graphs using clad*

```
RooGaussian::Translate(vector<string>&& params) const {
    return "Double_t" + getTempName() + "
        {" + getPrefix(params) +
        "const double arg = x - mean;
        const double sig = sigma;" +
        getTempName() + "= std::exp(-0.5*arg*arg/(sig*sig));
        }";
}
```

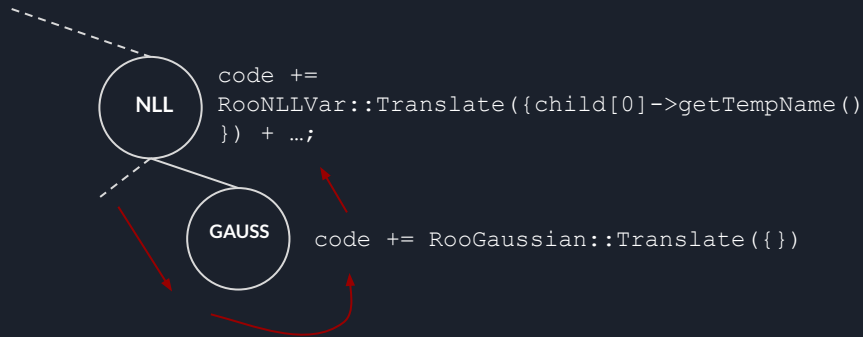
```
RooAutodiffGaussianNLL::getPrefix(
    vector<string>&& parameters){
    if(parameters.size() != 0)
        return "double x = " + parameters[0]
            + ", mean = " + parameters[1] +
            ", sigma = " + parameters[2] ";";
    else
        return "";
}
```

```
RooNLLVar::Translate(vector<string>&& params) const {
    return "double " + getTempName() + "
        {" + getPrefix(params) +
        "double nll = 0.0;
        // for (auto x : *data)
        nll += -std::log(x);" +
        getTempName() + "= nll;
        }";
}
```

```
RooNLLVar::getPrefix(vector<string>&& parameters) {
    if(parameters.size() != 0)
        return "double x = " + parameters[0] + ";";
    else
        return "";
}
```

# Possible Approaches Contd.

*Clad design : Approach 2 - Squash graphs using clad*



```
{
  double gauss_result_0;
  {
    const double arg = x - mean;
    const double sig = sigma;
    gauss_result_0 = std::exp(-0.5*arg*arg/(sig*sig));
  }
  double nll_result_0;
  {
    double x = gauss_result_0;
    double nll = 0.0;
    nll += -std::log(x);
    nll_result_0 = nll;
  }
}
```



## Expected Timeline of Work

For this month, I will be trying to figure out adding the clone class and running it on a basic example. If that works out, I will move to automatically generating the squashed code.





Thank you!

Any questions?