Extending TF1: Argument parsing, function composition, and vectorization

Arthur Tsang and Lorenzo Moneta
CERN
(Dated: September 13, 2017)

In this project, we extend the functionality of the TF1 function class in ROOT. We add argument parsing, making it possible to freely pass variables and parameters into pre-defined and user-defined functions. We also introduce a syntax to use certain compositions of functions, namely normalized sums and convolutions, directly in TF1. Finally, we introduce some simple vectorization functionality to TF1 and demonstrate the potential to speed up parallelizable computations.

I. INTRODUCTION

ROOT is an extensive C++ framework and the main tool used in particle physics data analysis. It features efficient data storage and retrieval, mathematical tools, curve fitting, minimization, plotting, data visualization, an interactive Cling C++ interpreter, and integration with Python and R. In this report, we describe how we extended a part of ROOT, the TF1 function class.

TF1, the focus of this project, is the class for mathematical functions in ROOT. It can be defined either with a pointer to a C++ function or with a formula given in a string. By default, TF1 is a one-dimensional function of the variable x, but there also exist TF2 and TF3, the two- and three-dimensional extensions which add variables y and z as appropriate. TF1 can also take arbitrary parameters, such as the mean, standard deviation, and amplitude in a gaussian, which in our formula syntax are always written in square brackets, whether the parameter is simply numbered or given a more descriptive name. These parameters can be set either manually or by fitting the TF1 to data.

We introduce three main features to TF1. These are: *argument parsing*, to plug new variables and parameters into existing functions, *function composition*, to create and use normalized sums and convolutions within a TF1 interface, and *vectorization* of variables, with the aim of increasing the efficiency of parallelizable computations. The first two already form part of the official ROOT repository on Github [1], and the vectorization is expected to arrive shortly after this report has been submitted.

II. ARGUMENT PARSING

Until now, in order to use an already-defined TF1 function with different parameters or variables, there would have been a couple of somewhat limited tricks available. The variable within a multi-dimensional function could be specified in square brackets immediately after the function name, e.g. `gaus[y]`. Furthermore, numbered parameters could be offset by specifying the new starting index in parentheses, e.g. `gaus[y](3)` to use parameters `[3]`, `[4]`, and `[5]`. However, these tricks only work for predefined functions and do not allow for more complicated replacement.

We introduce a more general argument parsing, which applies to all formula-based functions and enables arbitrary replacement including function nesting. It uses regular function application syntax, where the variables and parameters are the arguments. To take the previous example, `gaus[y](3)` can now be written more systematically as `gaus(y, [3], [4], [5])`. This more flexible syntax immediately allows for further variations thereon, such as `gaus(x+y, [Const], sin([beta]), [0])`.

To keep this syntax convenient, we introduce some more shortcuts. A list of parameters in ascending order can be given as `[`i`..`j`]` to start numbering at i and continue until j. Furthermore, if we only want to rename the parameters, the variables can be left implied, and if we want to change only variables, the parameters can be left implied. To avoid confusion, replacing a parameter with anything other than a single parameter requires all arguments to be explicitly given.

III. FUNCTION COMPOSITION

In this project, we concentrate on two types of function compositions: normalized sums and convolutions. Although there have already existed separate classes for these compositions, `TF1NormSum` and `TF1Convolution`, our contribution was to integrate their functionality directly into TF1, so that these composed functions are as easy to use as regular formula-based functions. Thus, we added the syntax `NORM(`$f_1, \ldots, f_n$`)` and `CONV(`f_1`,`f_2`)` to the TF1 formula string. Furthermore, any changes in the parameters or range of the TF1 are automatically relayed to the underlying composition class.

A. Normalized sums

A normalized sum is a weighted sum of functions, f_i, where the coefficients, c_i, are computed after normalizing each function:

$$\mathrm{NSUM}(f_1, \ldots, f_n)(\mathbf{x}, \mathbf{p}) = \sum_{i=1}^{n} c_i \cdot \frac{f_i(\mathbf{x}, \mathbf{p})}{\int f_i(\boldsymbol{\xi}, \mathbf{p}) d\boldsymbol{\xi}} \qquad (1)$$

If each f_i gives a distribution of events from a given process i, then normalization has the advantage that c_i now represents the count of these events. For example, take

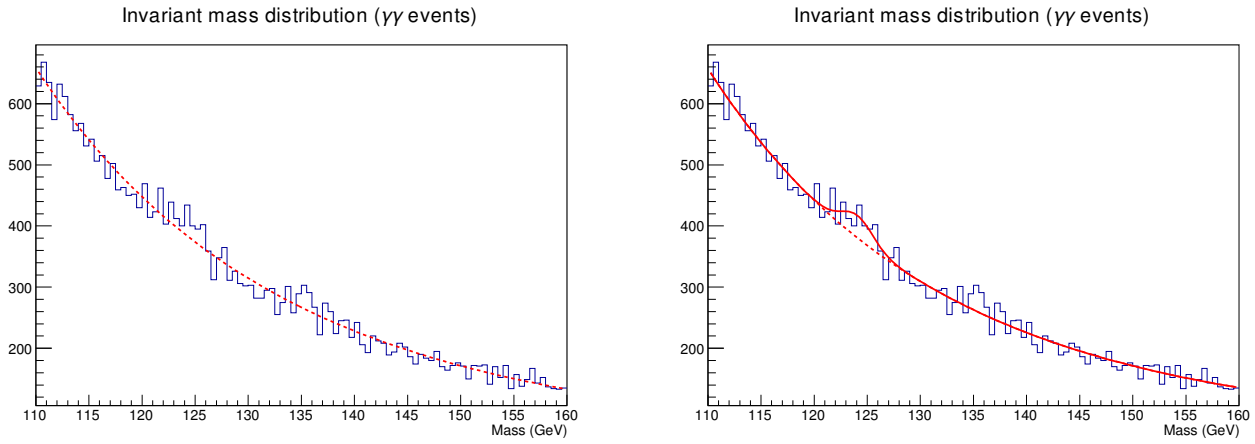

FIG. 1. This toy $\gamma\gamma$ dataset illustrates how NSUM can help to fit distributions composed of multiple processes. *Left:* A double exponential background fit to data. *Right:* A normalized sum composed of this background and a gaussian peak, fit again to data. The dotted line shows our fit with the gaussian bump removed.

.

$\gamma\gamma$ events, where the variable of interest, $\mathbf{x}$, is the invariant mass of the two photons. Since this is a decay product of the Higgs, we expect a narrow gaussian bump at the Higgs mass. Of course, this is in addition to a much larger background, which, it turns out, we can estimate as a double exponential, a function of the form $\exp(a_0x+a_1x^2)$. We can then define a normalized sum by combining this background with a gaussian Higgs bump, which we can fit to data as shown in Figure 1.

Of course, it was possible to do such an analysis before our work, but we provide a convenient way to define normalized sums directly in `TF1` with one line of code. In this example, once we define the double-exponential background, `expo2`, the normalized sum is just:

```
TF1("model", "NSUM(expo2, gaus)", 110, 160)
```

where the last two numbers are the upper and lower x bounds of our fitting region.

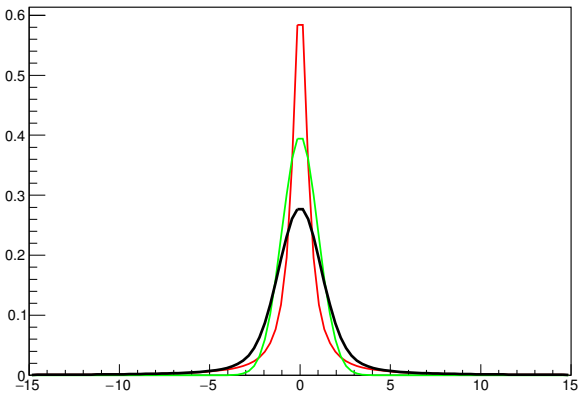

FIG. 2. The Voigt distribution, *black*, constructed with the `CONV(...)` syntax, is a convolution of the Breit-Wigner theoretical function, *red, taller*, and a gaussian resolution function, *green*.

B. Convolutions

Loosely speaking, a convolution of two functions is the result of "blurring" one by the other. More formally, the convolution of two functions, f and g, is defined as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(\xi)g(x-\xi)d\xi. \qquad (2)$$

A typical use case is where one function gives the theoretical distribution and the other a resolution function. An important caveat is that we can only evaluate a convolution accurately when both functions go near zero outside of the range of the convolution, since we integrate numerically with finite bounds. Figure 2 shows an example of a convolution using the `TF1`:

```
TF1("voigt", "CONV(breitwigner, gausn)",
    -15, 15)
```

where `breitwigner` and `gausn` are standard, pre-defined distributions.

IV. VECTORIZATION

The final part of this project is to enable a formula-based `TF1` to take vectorized inputs, in order to make parallelizable computations more efficient. The idea is that if we use a 128-bit register, a single operation can act at once on two 64-bit doubles, or four 32-bit floats, or other equivalent combinations. This principle is called SIMD (Single Instruction, Multiple Data), and the implementation which we use, SSE (Streaming SIMD Extensions), comes originally from Intel in 1999 [2].

Formula	Vectorized (s)	Unvectorized (s)
$x+1$	2.91(2)	4.71(8)
$x*x+x+1$	3.304(2)	4.71(8)
$x*x*x+x*x+x+1$	3.91(3)	4.78(7)
$x*x*x*x+x*x*x+x*x+x+1$	4.79(4)	4.81(7)
$x*x*x*x*x+x*x*x*x+x*x*x+x*x+x+1$	5.86(1)	4.86(8)

TABLE I. Using vectors of size 2, a for-loop of 100 million vectorized or 200 million unvectorized evaluations is timed for formulas of various complexities. We find that the vectorized calculation times depend strongly on the complexity of the formula, so while the vectorized mode is much faster for simple formulas, in complex formulas, the vectorization actually worsens performance.

While unvectorized evaluation remains the default, we introduce a method to toggle a vectorized mode in `TF1`, illustrated below:

```
f = new TF1("f", "2*x");
// Turn on vectorized mode
f->SetVectorized(true);
// Now we create a vectorized input
ROOT::Double_v x;
x[0] = 1; x[1] = 2;
// And test it
cout << f->Eval(x) << endl; // prints [2,4]
```

To measure the effect of vectorization, we time function evaluation on both vectorized and unvectorized inputs for formulas of varying complexity. The speed for vectorized input depends so strongly on function complexity, that while vectorization almost doubles the speed for simple formulas, it hurts performance for complex formulas, as shown in Table I. Thus, while one should be careful about when to use it, vectorization can still be a useful addition to `TF1`.

V. CONCLUSION

To summarize, we have introduced three main features to `TF1`: argument parsing, function composition, and vectorization. These new features add flexibility to TF1 and lay the foundations for optimizing parallelizable code. Since these contributions are all ready or almost ready for the next ROOT release, we hope to see thousands of ROOT users take advantage of what we have to offer in the near future.

VI. ACKNOWLEDGMENTS

I would like to thank the Summer Student Programme for giving me the fantastic opportunity to work on ROOT at CERN this summer, and I'd especially like to thank my supervisor, Lorenzo Moneta, for his constant help and support throughout the project.

[1] The official ROOT repository. URL `https://github.com/root-project/root`.

[2] NERSC (National Energy Research Scientific Computing Center). Vectorization, October 2016. URL `http://www.nersc.gov/users/accounts/user-accounts/acknowledge-nersc/`.