A Library of Function Objects

J. Boudreau, Mark Fischler, Petar Maksimovic

April 2, 2018

0.1 Why Function Objects?

In many applications it is desireable to treat mathematical functions as objects; the action of function-objects on their arguments and on each other (in other words, their algebra) can be defined in C++ so that it reflects the acutal mathematics, and instances of thes functions can be applied flexibly at run time either to data representing arugments or to other functions. Well-known use cases include: plotting, data modelling, simulation, function approximation, and integral transforms.

Using pointers-to-functions is a frequently used approach in either C or FORTRAN which gives some run-time flexibility but not nearly the power of function-objects. Using the native C math library we can write `sin(x)+exp(x)`, but we cannot write `sin+exp` nor pass this sum to other procedures. However, since we are free to overload operators in C++, we can get around this shortcoming by endowing the abstract interface to all function classes with all the operations we want our functions to have. For clarity we have attempted to restrict this interface to mathematically well defined operations, which will be discussed below. Another way to view this design is as one in which the abstract interface to functions permits the function library to be extended *not only through subclassing*, but also through "composition"[1]. We believe that the composition of functions through arithmetic operations is simple and intuitive since it is based on algebraic rules we've learned during childhood and is expressed in the same natural language.

In addition, we want to control the shape of a function: when we fit a function to data, for example. We can do so by possibly associating one or more *parameters* to a function, such as amplitude or frequency, lifetime, or width. This can be accomplished in C++ with parameter objects that can be part of or composed together with the functions. Altering a parameter alters the function or functions of which it is a component.

We have written a small class library ("GenericFunctions") which implements the features described above, for inclusion in the CLHEP project. Although our class library does not contain a comprehensive set of functions for mathematics and physics, it does provide an extensible framework for developing such a library. At the present writing it contains:

- An abstract base class for functions.
- Classes representing parameters.
- Arithmetic operations acting on both functions and parameters
- Class representing a possibly multidimensional *argument* to a function
- A small set of implemented functions

0.2 A Word to the Wise

Somebody told you once that C++ software is self-documenting. Being generally a trusting individual, you believed this for a while. But now, you're not so naive.

You're going to need documentation in order to make sense of the Generic Functions library. *This* is the documentation you'll need. There are a large number of classes here that you as a user don't need to know about at all. The header files aren't encrypted so you can browse them if you like, but you won't learn much that way. Read this documention instead. Thank you.

[1]here we mean composition in the sense of obect composition not function composition

0.3 Example Application

Our example application is a program to demonstrate the phenomena of interference and diffraction. This standalone program should allow one to control the width and separation of two slits in a filter, and also the intensity of light from each of the slits. As we change the parameters describing these variables we wish to see the *impulse function*, or the intensity of light radiation at the position of the filter, change in real time. Also we wish to simultaneously see the *response function* or the intensity pattern on the far screen, change in a way that is controlled by the same parameters. For this example, we are not going to worry about how to display the function. Graphics are outside the scope of the Generic Functions library. However just imagine that there is a plotter somewhere that gets a function object `f` and invokes the function-call operator during plotting, like this:

```
double y = f(x);  // f is a function object
```

The construction of functions is more involved than their invocation, so we're going to look at the code that sets up the functions and ties their shapes to the four parameters listed above. This code is shown in Example **??**, while screen shots from an application are shown in Figs. **??**, **??**, and **??**.

Fig. **??** shows both slits wide open and the classic double-slit interference pattern on the screen. Fig. **??** shows the the pattern when one of the slits is partially closed and the interference fringes are less sharp, and Fig **??** shows the one of the slits fully closed. In this last case you can see that the interference pattern has turned into a single-slit diffraction pattern.

We have two functions that need to be displayed: the impulse function and the response function. Neither of these functions are part of the library *per se*, but we can build them both out of the primitive functions `Rectangular` (for the impulse function) and `Sin` and `Cos` (for the response function). which are in the library. The response function, by the way, is given by the following expression:

$$I = [A_0 \sin ax/2/(ax/2)]^2 + [A_1 \sin ax/2/(ax/2)]^2 + 2A_0A_1 [\sin ax/2/(ax/2)]^2 \cos dx$$

where $x = \sin\theta$ and a is equal to the slit widht in units of the wavelength, d is equal to the separation between the slits in units of the wavelength, and A_0 and A_1 are the amplitudes from the two slits. The functions we require are simple enough to be built easily but complicated enough to illustrate several fundamental features of the library.

The basic parameters of the model are the intensities of the two slits, the width of the slits (this program does not allow the two slit widths to be varied independently) and the separation. These parameters are set up in lines 1-4 of the example. The variables corresponding to these parameters are called `a1`, `a2`, `s`, and `d`.

However some parameters of the impulse and response functions do not conveniently map onto these parameters but to simple combinations thereof. So, we can make derived parameters out of the basic input parameters. A derived parameter is a `GENPARAMETER`. This is shown in lines 5-8 where derived parameters `x0_0`, `x0_1`, `x1_0`, `x1_1` are defined in terms of input parameters.

The impulse function will be built out of two rectangular functions. So, we instantiate these functions (Line 9), connect their internal parameters to the input parameters (Line 10-11) and to the derived parameters (Lines 12-15). The input parameters are now referenced both by the derived parameters and by functions and **must not go out of scope** until the functions and derived parameters are no longer needed.

Now, whenever we vary the external parameter we're going to change the shape of the function. The two rectangular functions can be added to obtain the response function (Line 16). The sum of the two functions maintains its connections to the controlling parameters. The four parameters with variable names `a1`, `a2`, `s`, and `d` now not only control the two rectangular functions, but also their sum.

In the next few lines we build the response function, which is somewhat more involved. First we make instances of the functions we're going to use (Lines 17-20). Among these functions is a function `x` of class