A Library of Function Objects

J. Boudreau, Mark Fischler, Petar Maksimovic

May 16, 2001

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 1, while screen shots from an application are shown in Figs. 2, 3, and 4.

Fig. 2 shows both slits wide open and the classic double-slit interference pattern on the screen. Fig. 3 shows the the pattern when one of the slits is partially closed and the interference fringes are less sharp, and Fig 4 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

```
// Define the parameters:
Parameter i1                  ("Intensity",   1.0,   0.0,   1.0);
Parameter i2                  ("Intensity",   1.0,   0.0,   1.0);
Parameter s                   ("Width",       0.5,   0.0,   5.0);
Parameter d                   ("Separation",  1.0,   0.0,   5.0);

// Derived Parameters:
GENPARAMETER x0_0 = -d-s;
GENPARAMETER x1_0 = -d+s;
GENPARAMETER x0_1 =  d-s;
GENPARAMETER x1_1 =  d+s;

// Manufacture the impulse function out of two rectangular functions:
Rectangular rect0,rect1;
 rect0.height().connectFrom(&i1);
 rect1.height().connectFrom(&i2);
 rect0.x0().connectFrom(&x0_0);
 rect0.x1().connectFrom(&x1_0);
 rect1.x0().connectFrom(&x0_1);
 rect1.x1().connectFrom(&x1_1);
 GENFUNCTION impulse = rect0+rect1;

// Manufacture the response function:
 Variable x;
 Sin    sine;
 Cos    cosine;
 Square square;
 GENFUNCTION alpha = s*x*0.5;
 GENFUNCTION beta  = d*x;
 GENFUNCTION amplitude0 = i1*sine(alpha)/alpha;
 GENFUNCTION amplitude1 = i2*sine(alpha)/alpha;
 GENFUNCTION response=
   square(amplitude0) +
   square(amplitude1) +
   2*amplitude0*amplitude1*cosine(beta);
```

Figure 1: Example. Use of Generic Functions libary. See text for explanation.

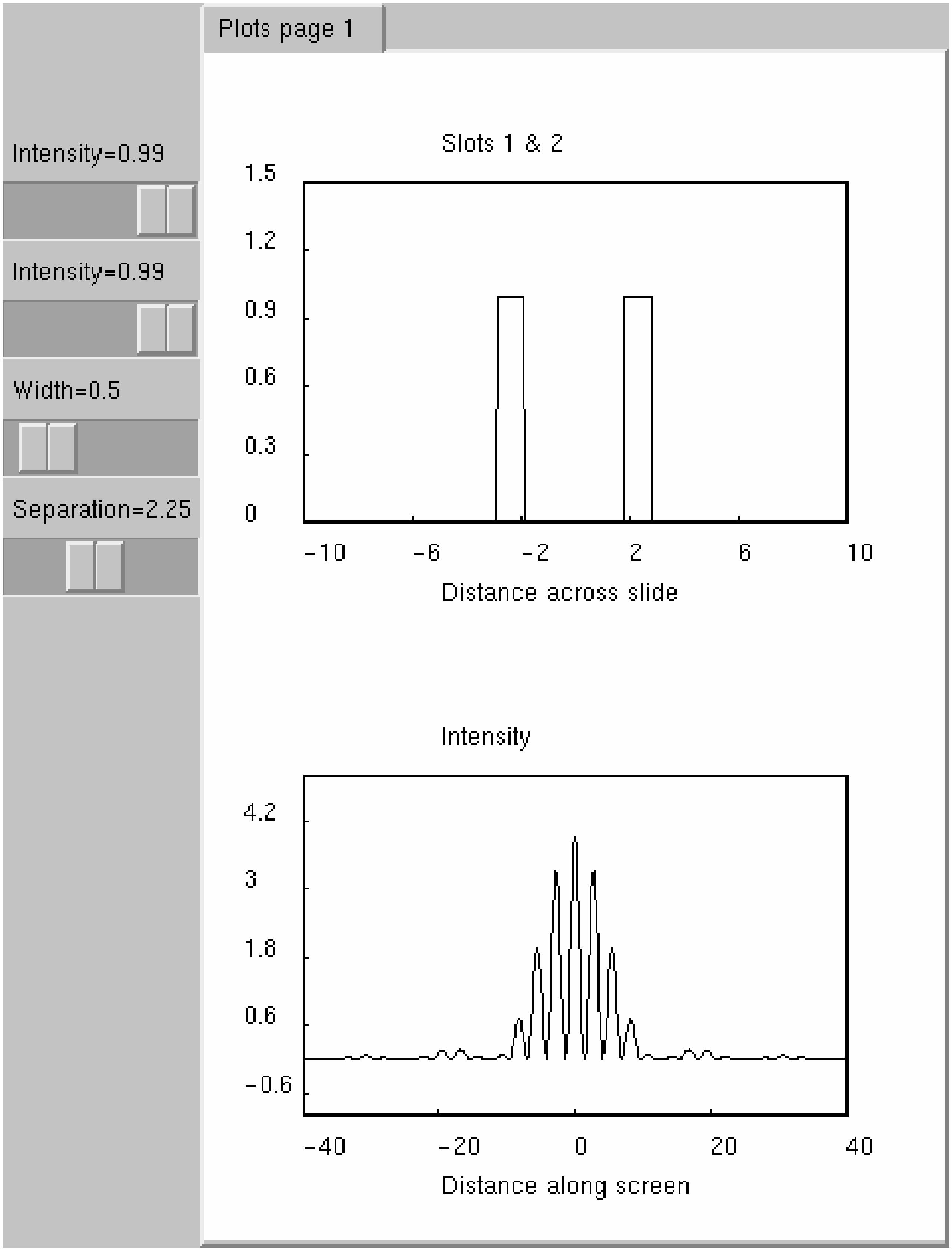

Figure 2: Picture of the example application which is discussed in the text. Above, the impulse function shows both slots wide open. Each of the sliders changes parameters and causes the plotter to update. Below, the response function shows the classic two-slit interference pattern.

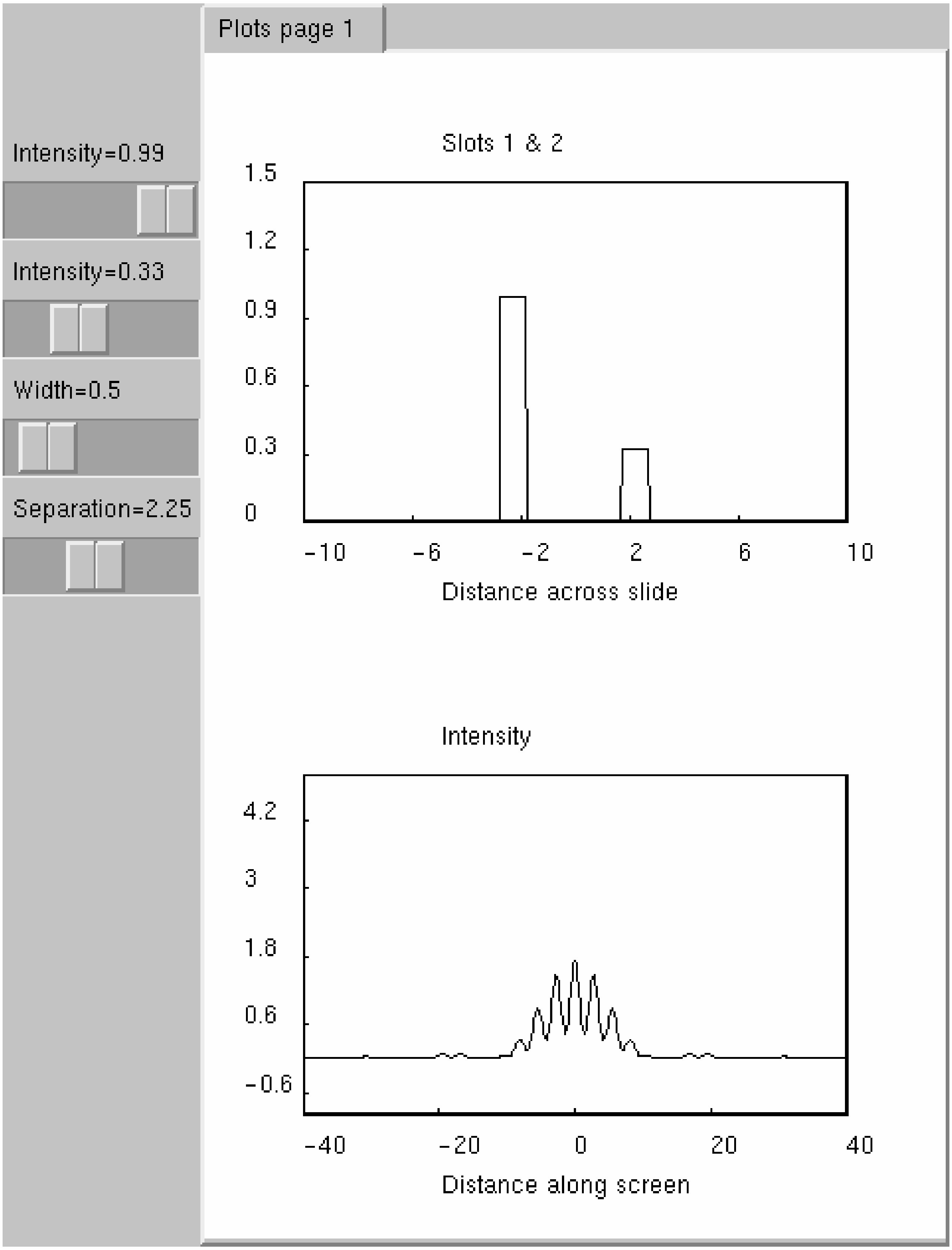

Figure 3: The parameter values are now changed. All of the functions, both primitive and derived, change their shape in response. Not that changing one parameter has affected both functions.

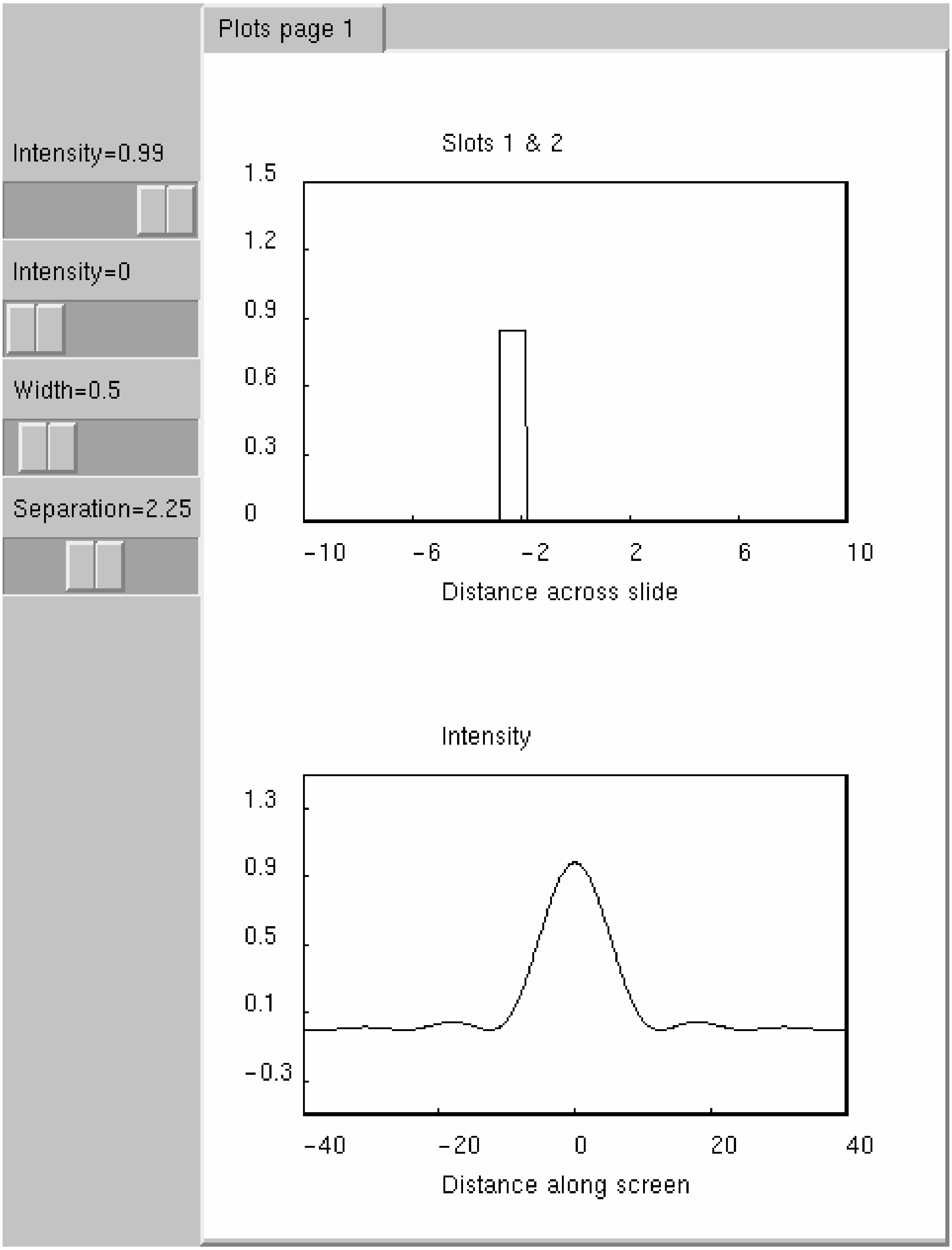

Figure 4: The second slit has been effectively closed, by setting the intensity of light through this slit to zero. The two-slit interference pattern then reduces to single-slit diffraction. This simple classroom demonstration program can now be used as an aid in discussing optics or quantum mechanics, and can be built in a few lines with a few minutes of programmer's time.

`Variable`, which just returns the value of its argument. We multiply this function by the constant 0.5 and by the Parameter `s` to obtain a new function, `alpha` in Line 21, and then perform a similar set of operations to obtain a function `beta` in Line 22. A derived function is a GENFUNCTION. Our algebra is defined on all of these data types. We also use functions of type `Square`, `Sin`, and `Cos`.

The composition of functions is indicated by parenthensis `()`. The function `square` acts by squaring its argument. The function `amplitude0` is a simple function of the angular distance `x`. So, the function which we naturally express as `square(amplitude0)` in Line 23, is the composition of the function `square` with the function `amplitude0`. In this example, function composition is used again in lines 24-28.

In general, we have tried to design this library so that single dimensional functions, multidimensional functions, double-precision constants and parameters all behave exactly as expected. If you find that they don't, then report it to us. It's a flaw that we want to fix. We have found no technical reason that functions, parameters, and constants cannot be made to mirror quite closely a mathematical language.

After the impulse function and the response function have been defined they can be passed around to other routines that may be written to the abstract interface of all functions, `AbsFunction`. This abstract interface allows one, essentially, to evaluate the function using `operator()(double x)` and to further compose it with functions, scalars, and parameters. In this example the functions `impulse` and `response` are passed on to some plotters. The shape of the function is controlled by the parameters; changing the vlaue of a parameter changes all the shapes of all functions that depend on it, both primitive and derived. In this example we have arranged things so that the parameters are modified when the scale is moved, giving users a way to control the shapes of both functions while seeing the visual representation of the function respond in real time to the scale setting.

0.4 Automatic derivatives

Any function, whether simple or composite, one dimensional or multidimensional, can return partial derivatives with respect to any argument. The computation is done symbolically, in general, not numerically. In case an analytic derivative is not available for any function, a numerical calculation is provided instead. For one-dimensional functions, the derivative is returned using this method:

```
GENFUNCTION fprime = f.prime(); // F is a GENFUNCTION
```

Partial derivatives of multidimensional functions with respect to one of their arguments is taken up in section 0.6 below. Information on whether the derivative of a particular function is analytic or not can be obtained through the use of the method `bool AbsFunction::hasAnalyticDerivative() const`.

0.5 The AbsFunction Class

The abstract base class for all functions of one or more dimensions is `AbsFunction`. The header file for this class also defines operations on the class. The base class describes the essential behaviour of all functions:

- virtual double operator() (double) const = 0
- virtual double operator() (const Argument &) const = 0;
- virtual unsigned int dimensionality() const;
- virtual Derivative prime() const;
- virtual bool hasAnalyticDerivative() const;

It also defines a default constructor and a virtual destructor, and hides the copy constructor and the default assignment operator. The data type `Derivative` is an `AbsFunction`, and may be handled like any other `AbsFunction`.

0.6 Multidimensional Functions and Arguments

Functions of more than one variable can be defined within this scheme. The way this is handled is using the class `Argument`, which essentially is a vector-like list of double-precision numbers. The argument is constructed using the integer number-of-dimensions; then each element of the argument can be set using `operator[](int i)`; for example:

```
Argument a(2);
a[0]=1.0;
a[1]=2.0;
cout << myFunction(a) << endl;
```

Users should take care to pass to functions an argument of the right dimensionality. Failure to do so will result in a run-time error. In addition care should be excercised when operating on functions with binary operations like addition or multiplication, that the functions both operands has the same dimensionality. Otherwise, a run-time error results.

In case of functions of a one-dimensional argument, one may either call

```
operator() (double) const
```

or

```
operator() (const Argument &) const,
```

and passing one-dimensional arguments only.

For multidimensional functions, one can obtain the partial derivative with respect to any argument using the method `virtual Derivative AbsFunction::partial( unsigned int i) const`. An alternate way to express this depends upon our ability to associate names to components of a multidimensional argument, and is described in the next section.

0.7 Variables

The class `Variable` is the most elementary function. The function value it returns is just the argument itself. It is useful because it can be used in function composition, like this:

```
Sin sin;
Variable x;
GENFUNCTION f = sin(2.0*x);
```

The class Variable can also be used to associate names with particular components of a multidimensional argument, using the alternate form of the constructor, like this:

```
Variable x(0), y(1);
Genfunction f = cos(y)sin(x) + sin(x)cos(y);
```

The function `f` is now a function of two variables. Alternately, the function may be constructed from the direct product operator `%`, see section 0.10 below.

If `v` is a variable and `f` is a multidimensional function, we can express the partial derivative of `f` with respect to the variable `v` like this:

```
GENFUNCTION fPartialV = f.partial(v);
```

i.e. using the method `virtual Derivative AbsFunction::partial( const Variable & v) const`.

0.8 AbsParameters

`AbsParameter` is an abstract base class for simple parameter classes representing double precision numbers. There are two main types of `AbsParameter`s. The first kind, simply called `Parameter`, comes with certain other services such as upper limits, lower limits, and connections to other parameters. The other kind is composite parameters, resulting from arithmetic operations on parameters. For now we will discuss only the first kind, `Parameter`. Neither class is really intended to be extended any further by users.

0.9 Parameters

The first service that parameters perform is to provide limits: it is impossible to set a parameter value outside of its limits–if this is attempted the parameter value will be set to the maximum or minimum allowed value.

Parameters can be made to take their value from other parameters. The programmer sets this relationship through the `connectFrom()` method, as illustrated here:

```
Rectangular rect;
Parameter height("Height", 1.0, 0.5, 2.0);
rect.height().connectFrom(height);
```

The first line creates a `Rectangular` function. The second line instantiates a parameter object called `height` with name "Height", lower and upper limits of 0.5 and 2 and a value of 1.0; The third line first accesses the internal parameter `height()` used by the function and connects this function permanently to the parameter `height`.

Functions allow access to their parameters. The shape of the function is then determined in one of three ways:

- The default value of the parameter can be taken.
- The value of the parameter can be set to a new value.
- The value of the parameter can be obtained from another parameter (if the parameter is connected).

When functions are used as arguments to arithmetic operations, and their parameters are connected, the connections are maintained in the composite function so created. However if a parameter in a function is connected after the function has been used in creating a composite, the composite function does not respond to changes in the value of the connected parameter. Instead it has the value of the parameter at the time it was created "frozen in.

0.9.1 Parameter class interface

Here is the public interface to the parameter class; the destructor, copy constructor , and assignment operator are not included on this list but are defined; the default constructor is not defined.

```
Constructors:

  Parameter(std::string,
            double,
            double x0=-1e100,
            double xl= 1e100);

  Accessors:

  const std::string & getName() const;
  double getValue() const;
  double getLowerLimit() const;
  double getUpperLimit() const;

  Modifiers:

  void setValue(double);
  void setLowerLimit(double lowerLimit);
  void setUpperLimit(double upperLimit);
  void connectFrom(Parameter *);

Print method:

std::ostream & operator << ( std::ostream &,
                             const Parameter &);
```

0.10 Arithmetic Operations

The result of binary or unary arithmetic operation involving an `AbsFunction` is another `AbsFunction`: however it is special kind of `AbsFunction`; generally the user will not care which kind since the actual return type depends on the last operation to be evaluated in the expression. The user needs to learn one trick, which is to handle the result of the arithmetic operation through the base class. This is demonstrated here where, we multiply an exponential with a cosine:

```
1  Exponential   exp;             // makes an exponential
2  Cos           cosine;          // make a cosine
3  AbsFunction & f = exp*cosine;  // multiply the functions.
4  cout << f(5) << endl;          // evaluate & print.
```

The third line of this example is standard and safe. It generates a function of class `FunctionProduct`, which also has type `AbsFunction`. All operations on functions return a value that has type `AbsFunction`, although the class depends on the actual operation, e.g. multiplication, division, et cetera. However the user does not care about which *class* is returned, but only cares that the return value has *type* `AbsFunction`, because he or she should handles it through the base class, `AbsFunction`.

Symbol	Name	Operand 1	Operand 2	return type
+,-,* /	Simple Arithmetic Operations	AbsFunction	AbsFunction	AbsFunction
		AbsFunction	Constant	AbsFunction
		Constant	AbsFunction	AbsFunction
		AbsFunction	AbsParameter	AbsFunction
		AbsParameter	AbsFunction	AbsFunction
		AbsParameter	AbsParameter	AbsParameter
		Constant	AbsParameter	AbsParameter
		AbsParameter	Constant	AbsParameter
unary -	Negation	AbsFunction		AbsFunction
		AbsParameter		AbsParameter
()	Composition	AbsFunction	AbsFunction	AbsFunction
		AbsFunction	AbsParameter	AbsParameter
%	Direct Product	AbsFunction	AbsFunction	AbsFunction
convolve	Convolution	AbsFunction	AbsFunction	AbsFunction

Another way of accomplishing the same thing is provided by the data type `GENFUNCTION`, typedef'd back to `const AbsFunction &`. It allows one to reexpress line 3 above in the following way:

```
GENFUNCTION f = exp*cos;
```

You can use this syntax if you find it more natural. For technically-minded people, the syntax generates a temporary object and a reference thereto. In ANSI standard C++ the temporary is guaranteed to persist until the refererence `f` goes out of scope.

Most of the operations that are valid on functions are also valid on parameters. Two parameters may be added, subtracted, multiplied, and divided-to yield another parameter. As in the case of functions, the return type of an operation yielding a parameter is an `AbsParameter &`, which is typedef'd as `GENPARAMETER`, for example:

```
GENPARAMETER x_s = Delta_m/Gamma;
```

where `Delta_m` and `Gamma` are both `AbsParameters`. Finally, operations between `AbsFunctions`, `AbsParameters`, and simple double precision numbers (we'll call them constants) are also defined. Unary negation is also defined on all of these data types.

Three additional operations are defined for functions. The direct product is for constructing multidimensional functions from lower-dimensional functions. For example a multivariate gaussian distribution of two uncorrelated variables could be built from two one dimensional gaussians using the following syntax:

```
Gaussian f;
Gaussian g;
GENFUNCTION h = f%g;
```

Mathematically, this is equivalent to $f(x) * g(y) = h(x, y)$. Some multidimensional functions may be constructed from lower dimensional functions but of course not all functions can be constructed this way. Where it fails, one can still resort to inheritance and build the needed function by subclassing `AbsFunction`. The picture in Fig. 5 illustrates a multidimensional function obtained by forming a three dimensional function from lower dimensional functions, the illustration is the probability density function for a higher excited state of hydrogen.

The composition operator is for taking functions of functions, or functions of parameters. It is denoted with the function call operator, `()`. For example the following syntax is valid:

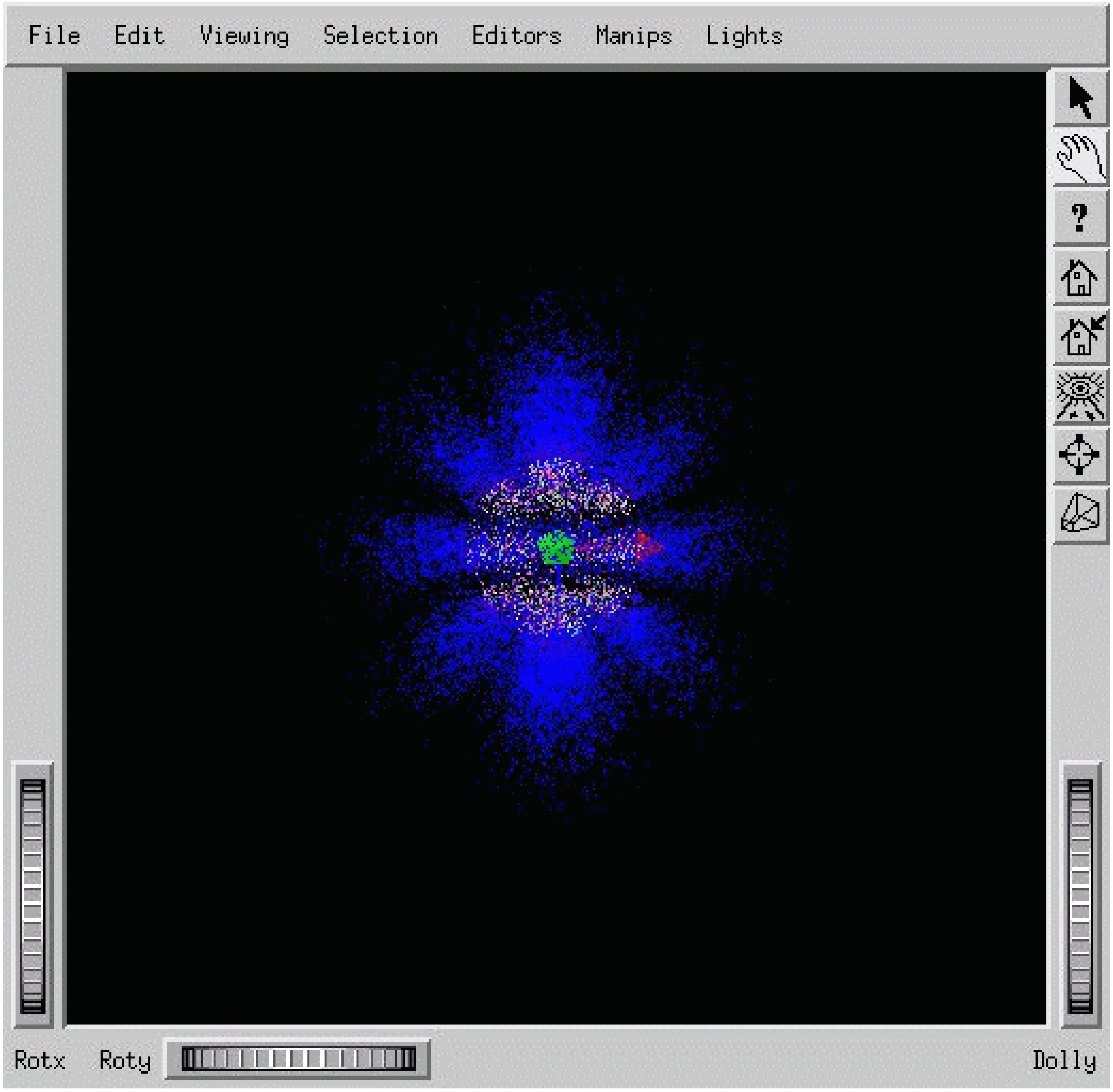

Figure 5: Multidimensional functions can be expressed as a direct product of lower dimensional functions; this example combines a Legendre Polynomial with the product of an exponential, a power function and a Laguerre Polynomial to obtain the probability density function of an electron in an excited state of hydrogen. The dots are generated by performing a random throwaway against a generic function of three dimensions.

```
Exponential exponential;
Sin sine;
GENFUNCTION f = sine(exponential) ;
```

A convolution of two functions has no special symbol-we ran out of them- but is produced with the convolve method. This function performs a numerical convolution of the two functions treating the first function as a response function, the second as a resolution function. The function has two addtional arguments, which are the limits of the numerical convolution. The convolve function is then used like this:

```
Exponential exp;
Gaussian      gauss;
GENFUNCTION h = convolve(exp, gauss, -10.0, 10.0);
```

The technique employed is to sample the product $f(x-y)g(y)$ at 200 points between $x0 < y < x1$, where f is the response function and g is the resolution function. We hope to use Fourier techniques to improve this method in due time.

0.11 Some Functions in the library

Several types of functions have been implemented in this scheme. A table of those functions is shown here. Certain functions, such as Legendre Polynomials, take arguments to their constructor that specify the order of the polynomial. In other cases internal parameters govern the shape of the function. The distinction is arbitrary at times, since, for example, Bessel functions need not be of integral order. In general we have preferred to leave parameters out of functions if one can obtain the same result by other means, for example, objects of class `Sin` do not have a frequency parameter since one can obtain this as follows:

```
Parameter freq (``freq'', 10.0);
Sin sine;
Variable x;
GENFUNCTION f = sine(freq*x);
```

0.12 Parameter to Argument Adaptors

Parameters control the shape of a function, which then act in the space of their arguments. In many applications it is useful to be able to create a new function out of an existing function by treating one of its parameters as an argument.

A use case would be the following. Suppose that one has written a function to describe a measured quantity, like lifetime, smeared with a resolution function which is a Gaussian. The resulting function is parametrized by σ, the width of the Gaussian distribution. However to use this function in an unbinned log likelihood fit, we often need to incorporate event-by-event estimates of σ. To do this is to formally promote the parameter σ to an argument, in other words to turn:

$$F(\alpha_i, \sigma; x) \tag{1}$$

into

$$F(\alpha_i; \sigma, x) \tag{2}$$

We do this using a helper class called `ParamToArgAdaptor`. The class is an `AbsFunction` that takes as arguments to its constructor:

Function	Name	Parameters
Variable	Returns its own input	
FixedConstant	Returns a constant	
Sqrt	Sqrt	
Square	Returns square of input	
Power	Returns a power of input	
Exp	Exponential	
Sin	Sine	
Cos	Cosine	
Tan	Tangent	
Ln	Natural Logarithm	
Erf	Error function	
ForwardExp	Forward Exponentail tail	decayConstant
ReverseExp	Reverse Exponential tail	decayConstant
LogGamma	Natural log of Gamma function	
IncompleteGamma	Incomplete Gamma Function	a
CumulativeChiSquare	Probability(χ^2)	
Gauss	Gaussian (Normal) distribution	mean sigma
Landau	Landau distribution	peak width
Rectangular	Rectangular function	x0 x1 baseline height
PeriodicRectangular	Periodic rectangular	spacing width height
SphericalBessel	Spherical Bessel Functions	
SphericalNeumann	Spherical Neumann Function	
AssociatedLaguerre	Associated Laguerre Polynomial	
AssociatedLegendre	Associated Legendre Polynomial	
AnalyticConvolution	Moser-Roussarie convolutions	frequency lifetime resolution
IntegralOrder::Bessel	Bessel and Neumann functions	
FractionalOrder::Bessel	Bessel and Neumann functions	order
BivariateGaussian	Gaussian in 2 variables	mean0, mean1, sigma0,sigma1, corr01
TrivariateGaussian	Gaussian in 3 variables	mean[0-2],sigma[0-2] corr[0-2][0-2]

- The name of an auxilary function, whose parameter it is to promote.
- The name of a member function of the auxilary function, which accesses the parameter to promote.

For example, one can turn a smeared expontial function of one variable into a function of two variables, where the second variable is the event-per-event error estimate.

```
Genfun::AnalyticConvolution  smearedExponential (Genfun::AnalyticConvolution::SMEARED_EXP);
ParamToArgAdaptor<AnalyticConvolution> smExpEventByEvent(smearedExponential,
                                         &AnalyticConvolution::sigma);
```

The function `smearedExponential` is a function of one argument, so the new function `smExpEventByEvent` is a function of 2 arguments. The last argument is the event-by-event resolution. Scale factors may be added to the newly created function, so that the actual value of the parameter used is some scale factor times the argument provided. The scale factor is in fact a parameter of the adaptor class, accessed via the method `scaleFactor`, and may be set or connected just like any other parameter.

In case two parameters need to be promoted along these lines, we provide an additional class:

```
DoubleParamToArgAdaptor
```

which differs from `ParamToArgAdaptor` only by the presence of an additional argument to the constructor (the name of a member function accessing the second parameter to be promoted) and the presence of a second parameter (an additional scale factor).

0.13 How to extend the Generic Functions package with your own function

Creating a function for the Generic Function package or inserting an existing function into the framework has low work overhead. Of course if the function is complicated and hard to write, this the framework does not make it easier! However there are only a small number of steps involved in creating a one- or multi-dimensional function and endowing it with parameters that control its shape. We present this procedure as a checklist:

1. Derive your function publically as a subclass of `AbsFunction`.
2. What is the dimensionality of your function? If the answer is "one", then you don't need to override the method

   ```
   virtual unsigned int dimensionality() const;
   ```

   because the base class provides a default implementation which returns the value 1. Otherwise, if your function has two or more dimensions, you will need to override the method.
3. Which parameters (if any) does your function depend on? For each parameter you need to add a `Parameter` data member for that parameter, and initialize it with its default value and range in the constructor. Also you will want a way of getting a reference to the parameter-and actually you will want two methods, overloaded on const. The purpose of having both methods is so that you will get back a parameter that you can modify if the function itself is modifiable, and a read-only parameter if the function is readonly. If overloading on const is strange to you, you can read up on it in Meyers; or you can just adapt both of these example lines:

```
Parameter & parm();
const Parameter & parm() const;
```

which shows, for example how to retrieve a parameter called "parm" from a function. This is important and the compiler errors may appear very mysterious if you don't write your parameter accessors this way.

4. You must provide an implementation of these two functions which are pure abstract in the base class:

   ```
   virtual double operator() (double) const;
   virtual double operator() (const Argument &) const;
   ```

   These operators are what your function "does" so how you go about doing this is your business. But typically, for one dimensional functions the second form of the function just calls the first; while for multidimensional functions the first form generates a run-time error and the second form checks the dimension of the argument before evaluating it.

5. The composition operator, `operator()( const AbsFunction &)` is overloaded in the base class and you don't want to hide it in the subclass, so put the line `using AbsFunction::operator()` in the header file of your derived class.

6. Copy constructors are required for the proper operation of your function within the framework. You can use the compiler-generated copy of this constructor if you wish; the usual caveats about dynamically allocated memory apply here as with any class. The assignment operator is confusing because it only allows one to assign Gaussians to Gaussians, exponentials to exponentials, et cetera, so this function should be turned off. The way to do this is to a) declare the assignment operator, b) make it private, and c) do not provide an implementation.

7. You will need to declare and define the method:

   ```
   virtual SubClass  * clone() const;
   ```

   where `SubClass` is your new class. This construction uses the covariant return type mechanism, since the function in the base class returns an `AbsFunction *`. The purpose of the routine is of course to return a pointer to a newly allocated object. The easiest way of implementing this function is to use the copy constructor that you wrote (if you wrote one, otherwise you can take the default copy constructor that the compiler wrote for you).

8. **Optional** If you will be providing an analytic derivative for your new function, then override the method

   ```
   Derivative partial (unsigned int) const;
   ```

   The function `Sin` provides a good example of how to do this. Should you choose to provide an analytic derivative, you should also then override the method

   ```
   bool hasAnalyticDerivative() const
   ```

   so that it returns true. Our preference is to put the implementation right in the header file since it provides useful information ("yes, this class has an analtyic derivative") for users.

0.14 Cut classes

GenericFunctions is based on our ability, in C++, to write classes having the algebra of real functions of one or more variables. The same techniques which have been applied here can be used to describe other algebras. In particular, there is an extremely useful extension to *cut classes*, which are objects that return a true or false decision on some data type, and which support the algebra of boolean operations. A set of base classes has been written to allow one to create cuts on any data type, to combine them using the operators `||`, `&&` and! We have written these classes to interoperate with with the standard template library.

A simple example will show the usefulness of this. In this example, the cut-object `IsPrime` and `IsInRange` have been written as subclasses of `Cut<int>`. In the example we use these cuts to select prime numbers in the range 30-60, and print out the result (31,37, 41, 43, 47, 53, 59) to the terminal screen:

```
int main(int, char **) {
  //
  // Make an array of integers:
  //
  const int LENGTH=100;
  int integers[LENGTH];
  //
  // Fill them with the integers:
  //
  for (int i=0;i<LENGTH;i++) integers[i]=i;
  //
  // Make an output iterator:
  //
  std::ostream_iterator<int> dest(std::cout,"\n");
  //
  // Cut on prime numbers between 30 and 60:
  //
  const Cut<int>::Predicate cut = IsPrime() && IsInRange(30, 60);
  std::remove_copy_if(integers, integers+LENGTH, dest, !cut);
  //
  // Bye:
  //
  return 0;
}
```

Let's look in detail at how one of these cut objects was declared (note that it inherits from `Cut<int>`):

```
class IsPrime:public Cut<int> {

public:

  // Constructor:
  IsPrime();

  // Destructor:
  virtual ~IsPrime();

  // Truth operator:
  bool operator () (const int & arg) const;
```

```
  // Clone
  virtual IsPrime *clone() const;

};
```

Readers familiar with the standard template library will ask, “How does this relate to STL predicate types? What has been gained, and what has been the cost?” One has gained the ability to combine existing cut-objects with boolean operations. The cost has been: some virtual function calls in the evaluation of these objects, and the necessity of inheriting from `Cut<Type>`, meaning also that two pure virtual functions:

- `virtual Cut * clone() const = 0;`
- `virtual bool operator ()( const Type & t ) const = 0;`

will have to be defined by the user.

Unlike the function-objects, we do not provide any concrete cut-objects. Instead what we provide is the base classes that allow one to easily write STL predicates supporting boolean operations[2] To access these, include the header file `CLHEP/GenericFunctions/TrackCutBase.hh`.

[2]To be sure, the STL has a set of classes that allow one to combine cuts using boolean operations, but their interface is appalling.