
MATHLIB 0.0.3 Review

Walter E. Brown
Marc Paterno

Table of Contents

1	Introduction	1
1.1	Mission and scope of this review	1
1.2	General impressions	2
2	Physical Organization	2
3	Build system	2
4	General Programming Practices	2
4.1	Design documentation	2
4.2	Need for <code>explicit</code>	3
4.3	Consistent naming	3
4.4	Resource management	3
4.5	Clear purpose for each class	3
4.6	Invariants	4
4.7	Forward declarations	4
4.8	Inclusion of headers	4
4.9	Namespaces	4
4.10	Feature testing	4
5	Detailed Analysis of Selected Packages	5
5.1	Special functions	5
5.2	General functions	5
6	Brief Analysis of Selected Packages	12
6.1	Integration	12
6.2	Differentiation	13
7	Conclusion	14

1 Introduction

1.1 Mission and scope of this review

We have been asked, on behalf of CMS, to review **MATHLIB** version 0.0.3. We address issues of *design* and *implementation*, utilizing such criteria as:

- maintainability,
- extensibility, and
- adherence to "best practices."

We do not address *coverage*, because we recognize that this version of **MATHLIB** is a work in progress.

In order to produce a timely result, we have restricted our detailed attention to:

- the special functions implementations (`SpecFunc`), and
- the hierarchy rooted at `IGenFunction` (`CxxFunc`).

We have given a somewhat less-detailed look to the other parts of the library, but believe our remarks below are representative of the library in its entirety.

1.2 General impressions

In general, we find that the library makes appropriate use of both procedural and object-oriented C++.

However, there seems to be a tendency to view library components as a sequence of independently-operating parts. We would recommend a more cohesive perspective in which the library is viewed as a whole and in which individual components are intended to provide functionality that can also be employed in other parts of the library. We are strong advocates of the principle that "each programming unit (class, function, etc.) has one purpose"; many of our recommendations below will lead directly in this direction.

2 Physical Organization

We believe the directory structure is quite appropriate. We found the hierarchy straightforward to comprehend and to navigate.

3 Build system

It is unclear what advantage is obtained by using Python as the scripting language to drive the building of GSL and then MathCore. Since the build system depends on a `bash`-compatible shell, it would seem a `bash` script would make fewer demands upon the user, who may not have Python available.

The script `build.py` seems to allow only in-source build. We strongly prefer out-of-source builds. Also, there seems to be no way to prevent the *install* from being done. We strongly prefer to *configure*, then *build*, then *test*, and only *install* after testing.

The direct use of the `configure` scripts seems more convenient. They do allow out-of-source builds, and also allow the user to control the time of installation.

We find the *test* facility inconvenient. It seems to support only in-source building of tests, requires manual (as opposed to automated) running of the tests, and gives no report of success or failure. We strongly prefer an out-of-source build, with automatic running of tests, and automatic analysis of the results -- providing the user with a clear and succinct statement of success or failure.

We note in passing that the static libraries build in the Windows/Cygwin environment, but the dynamic libraries do not. Fixing this may be worth pursuing.

4 General Programming Practices

4.1 Design documentation

We note the apparent lack of a *design document* (i.e., a document that describes the design of each class, and which gives a coherent overview of subsystems). This lack has hindered our efforts to understand how some of the parts of the library are intended to operate alone and to co-operate with other components in accomplishing user tasks. We believe that the lack of a design document is likely to lead to lack of coherence in the design, and will make maintenance more difficult.

We suggest the creation of a "roadmap" document, describing the purpose of the library and of each module, and describing important features of the design of each class. This should be a working document -- one kept up-to-date as the library grows.

4.2 Need for `explicit`

There are several classes with single-argument constructors that could give rise to unexpected conversions. Single-argument constructors are best declared `explicit`, unless it really does make sense, under all circumstances, for a conversion to happen. (This recommendation applies equally to multi-argument constructors with default values.)

4.3 Consistent naming

We have found a number of code instances in which a common concept is spelled in a variety of ways. Not only is there inconsistency in spelling, but also in the use of singular versus plural forms, and in the lengths of the abbreviations used. We have found, for example:

- `parameter`
- `Parameters`
- `par`
- `param`

and so on. We recommend, instead, the uniform use of a single, consistent, spelling in all contexts that name a single concept or entity. Not only is there less of a mental burden on developers and users (i.e., less to be remembered), it also reduces the likelihood of error -- we have found at least one case (see `ParamFunction`, below) in which an inadvertant difference in spelling led to failure to override a virtual function, and instead resulted in introduction of an unrelated non-virtual function. Consistent naming would have helped to avoid this error.

4.4 Resource management

We have found instances of native (*bare*) pointer use that will lead to resource management errors in case of exceptions, errors, or other corner cases. We prefer to use a managed pointer template to avoid the use of use of bare pointers holding owned items.

Not only do well-designed and -documented *smart pointers* (managed pointer templates) exist, their use has been shown to address these and related issues with little or no need for explicit coding beyond the correct use of the template.

We have, in this document, refrained from introducing any managed pointers in our code snippets. We did so in the interest of clarity, to emphasize the code's structure. In actual implementation, however, we would give very strong consideration to the use of managed pointers wherever bare pointers are found.

4.5 Clear purpose for each class

Some classes have terse, cryptic names (e.g. `CParamFunction`). Especially when class names may not be clear, we believe it is important that the header for each class contain brief documentation explaining the *purpose* of the class.

If it is difficult to produce such a one- or two-sentence description, it is likely, in our experience, that the class lacks cohesiveness.

4.6 Invariants

The purpose of some classes' member functions is unclear, and some implementations seem very complex, because of absent or unclear class and function invariants. We recommend that each entity be documented with the invariant it establishes and maintains.

In general, we hold to the principle that objects should be immediately usable upon construction. We strongly prefer that users not be required to construct and later configure an object (so-called *two-phase initialization*).

4.7 Forward declarations

We strongly recommend *against* the practice that clients of a module forward-declare any of that module's classes. Instead, when forward declarations are considered desirable, we urge the provider of the module being used create a header file, suitable for client code to `#include` and which internally forward-declares the appropriate name(s). (For example, the standard library provides `iosfwd` for such a purpose.)

The advantage of the additional level of indirection is increased maintainability. For example, this allows the implementer the latitude to rename the underlying class, to add template parameters with default values, or to change a class to a specialization of a template.

4.8 Inclusion of headers

We find that each of the source (.cpp) files in the subpackages *SpecFunc* and *StatFunc* fails to include its respective headers. (Not all subpackages share this failure.)

We strongly urge that *every* source file that defines a function should include the header that declares that function, unless the function is intended only to be used internally by that source file. (In our experience, the latter situation is relatively uncommon.)

4.9 Namespaces

Other than `gsl` code, We note that all the names defined in the **MATHLIB** product reside in a single namespace: `mathlib`. Some "implementation details" reside in nested namespaces. We support the practice of placing implementation details in nested namespaces.

We anticipate that, in the future, vendors will supply special functions in their implementations of the Standard Library. To facilitate the future transition to use of these functions, we recommend that the functions proposed for the Standard Library be separated into their own namespace. We recommend any of the following names for this namespace: `stdmathlib`, `trlmathlib`, or `specfunc`.

This will allow future use of a namespace `alias` in the `SpecFunc.h` header, with no need to modify client code:

```
// This is the entirety of the future SpecFunc.h:
#include <cmath>
namespace stdmathlib = std;
```

Of course, client code using `SpecFunc.h` would require re-compilation.

4.10 Feature testing

We have noted several base classes that provide optional functionality, and so require users to inquire

whether a given instance of such a class provides that optional functionality. We believe this is a poor design practice, as it leads to user code littered with functionality testing.

In an object-oriented design, it is generally considered a preferred practice to make use of inheritance of interfaces (i.e., multiple inheritance) in such a way as to avoid the need for feature testing.

5 Detailed Analysis of Selected Packages

In this section, we consider in greater detail two of the packages in **MATHLIB**. We have selected `SpecFunc`, because it is in a sense the "lowest level" of functionality provided. We have selected `Cp-pFunc` because it seems to form the core of the functionality of **MATHLIB** -- several of the other packages manipulate instances of the classes found in `Cp-pFunc`.

5.1 Special functions

We note that the proposed standard interfaces for these functions have been respected. This will facilitate future transitions.

We note that only the `double` implementations, and not the `float` or `long double`, are provided for the various functions. We regard this as a sensible first step, likely to suffice until implementations conforming to the proposed standard are available from vendors.

However, it seems that the error-reporting facility of the special functions does *not* meet the proposed standard. We recommend that, in the near future, these functions' implementations be revisited so as to provide error-handling consistent with that specified by the draft TR1.

An implementation of the special functions conforming to this draft can return a *NaN* to indicate a domain error. We suggest that the wrappers in `SpecFunc` do so. An example implementation is:

```
double dNaN() { /* defined to return a NaN of type double */ }

double cyl_bessel_j(double nu, double x) {
    if ( x < 0.0 ) return dNaN() + 1.0; // See Notes 1 & 2
    if ( nu >= 128.0 ) return dNaN() + 1.0; // See Notes 2 & 3
    double result = gs_sf_bessel_Jnu(nu, x);
    return ( /* GSL produced an error */ )
        ? dNaN() + 1.0 // See Note 2
        : result;
}
```

Note 1: The draft TR1 requires that negative values of *x* be excluded from the domain of `cyl_bessel_j`.

Note 2: The construction `dNaN() + 1.0` is suggested so that experiments (or users) who use the common floating-point processor facilities to mask floating-point exceptions can either (1) allow the propagation of NaNs without a floating-point exception, or (2) cause a floating-point exception at the point of the addition.

Note 3: We recommend the test on the value of *nu* for portability. Evaluation of the function for larger values of *nu* yields "implementation-defined" behavior, and hence is inherently non-portable.

5.2 General functions

These functions are implemented in the module `Cp-pFunc`. As previously noted, these classes seem not to provide documentation of their intended purpose. We have therefore synthesized the apparent purpose

of several of these classes, based on their declarations and the behaviors these declarations permit.

- `IGenFunction`: an interface that represents a function of one argument, and provides for cloning capability. It also provides gradient calculation, but we have discounted this since its default implementation (!) would give the wrong answer for most functions.
- `IParamFunction`: an interface representing a function of one argument and which function also has zero or more bindable parameters. Via inheritance, it also presents the interface of `IGenFunction`.
- `ParamFunction`: an abstract base class that provides partial implementation of the `IParamFunction` interface.
- `CParamFunction`: a concrete class wrapping a free functions of a particular signature, `(double, std::vector<double> const&)`. It implements the remainder of the `IParamFunction` interface.
- `Polynomial`: represents a polynomial, presenting the `IParamFunction` interface.

We are uncertain that we have accurately captured the intended purposes of these classes.

We will later (A proposed refactoring) propose a modification of this hierarchy. First, we briefly discuss some members of the hierarchy.

5.2.1 `IGenFunction`

This class is the root of the inheritance hierarchy. It is an *abstract base class* because it has pure virtual members. But it is *not* a pure interface, because some of the virtual members have implementations. In addition, these implementations are inlined.

The `inline` definition of virtual members is of dubious value. The functions will rarely be `inline`-d by the compiler, and never when used in the intended polymorphic fashion.

We also note the presence of the feature-testing function `providesGradient`.

5.2.2 `IParamFunction`

We note the presence of the feature-testing function ``providesParamGradient`.

5.2.3 `ParamFunction`

This class includes the function `providesParameterGradient`, which seems likely to be an accidental misspelling of `providesParamGradient` declared in its `IParamFunction` base. This highlights the importance of *consistent naming*, as noted above. The function `providesParameterGradient` is *not* virtual, and `ParamFunction` retains the `IParamFunction` default implementation of `providesParamGradient`.

5.2.4 A proposed refactoring

We suggest a modification of the design so as to follow the *non-virtual interface* ("NVI") principle. We believe such an approach is applicable and beneficial for several reasons:

- Some functions (such as `operator()(double, std::vector<double> const&)` in `ParamFunction`) are clearly expressions of a non-virtual interface; there is no reason for any derived class to implement this function differently.

- Other functions (such as `setParameters(std::vector<double> const&)` in `IParamFunction`) would benefit from enforcement of class invariants for each subclass, which can be provided automatically by the base class.

Furthermore, using NVI allows the the interfaces in the hierarchy to take their "natural form" for users, while providing a distinct interface for specialization to *implementers* of subclasses.

Finally, the redesign allows for more flexibility. For example, it is straightforward under our proposed design to provide tools which would allow the user to wrap any of the functions of `SpecFunc` in a class that provides our equivalent of the `IGenFunction` interface -- and thus would allow their use with `Integration`, `Chebyshev`, etc.

The following code samples sketch our proposed solution; we have shown code inline for exposition purposes only.

```
// File IGeneralFunction.h
#ifndef MATHCORE_IGENERALFUNCTION_H
#define MATHCORE_IGENERALFUNCTION_H

class IGeneralFunction // IGF
{
    // Interface for functions of one argument.

public:
    virtual ~IGeneralFunction() { /**/ }

    virtual IGeneralFunction * clone() const = 0;

    double operator()( double x ) const { return do_evaluateFunction(x); }

private:
    virtual double do_evaluateFunction( double ) const = 0;
}; // IGF

#endif // MATHCORE_IGENERALFUNCTION_H


// File IGradient.h
#ifndef MATHCORE_IGRADIANT_H
#define MATHCORE_IGRADIANT_H

class IGradient
{
    // Interface for entities that supply a gradient.

public:
    virtual ~IGradient() { /**/ }

    double gradient( double x ) const { return do_gradient(x); }

private:
    virtual double do_gradient( double ) const = 0;
}; // IGradient

#endif // MATHCORE_IGRADIANT_H
```

```

// File IParameterizedFunction.h
#ifndef MATHCORE_IPARAMETERIZEDFUNCTION_H
#define MATHCORE_IPARAMETERIZEDFUNCTION_H

#include "IGeneralFunction.h"
#include <vector>

typedef std::vector<double> dblvec; // exposition only

class IParameterizedFunction // IPF
: public virtual IGeneralFunction
{
    // Interface for single-argument functions that have
    // a fixed number of bindable parameters.

public:
    virtual ~IParameterizedFunction() { /**/ }

    virtual IParameterizedFunction * clone() const = 0;

    void changeParameters( dblvec const & p ) { do_changeParameters(p); }

    size_t numberOfParameters() const { return do_numberOfParameters(); }

    void fetchParameters( dblvec & p ) const { return do_fetchParameters(p); }

    using IGeneralFunction::operator();
    double operator() ( double x, dblvec const & p ) {
        changeParameters(p);
        return operator()(x);
    }

private:
    virtual void do_changeParameters( dblvec const & ) = 0;
    virtual size_t do_numberOfParameters() const = 0;
    virtual void do_fetchParameters( dblvec & ) const = 0;
    virtual double do_evaluateFunction( double ) const = 0;
}; // IPF

// Notes:
//
// 1. In a class inheriting this interface, the changeParameters
//    and do_changeParameters functions are permitted to throw
//    an exception (e.g., functionConfigurationError) if the
//    supplied dblvec's size is inappropriate.
//
// 2. If it is needed to support an interface to single-argument
//    functions that have a variable number of bindable parameters,
//    we recommend duplicating this class's interface and adjusting
//    its public implementations so as to support this behavior.

#endif // MATHCORE_IPARAMETERIZEDFUNCTION_H

// File IGeneralFunctionWithGradient.h
#ifndef MATHCORE_IGENERALFUNCTIONWITHGRADIENT_H
#define MATHCORE_IGENERALFUNCTIONWITHGRADIENT_H

#include "IGeneralFunction.h"
#include "IGradient.h"

```



```

class IGeneralFunctionWithGradient // IGFwG
: public virtual IGeneralFunction
, public IGradient
{
    // Interface for single-argument functions that also supply a gradient.

public:
    virtual ~IGeneralFunctionWithGradient() { /**/ }

    virtual IGeneralFunctionWithGradient * clone() const = 0;

    void fdf( double x, double & f, double & df ) const {
        f = operator()(x);
        df = gradient (x);
    }

private:
    virtual double do_evaluateFunction( double ) const = 0;
    virtual double do_gradient( double ) const = 0;

}; // IGFwG

#endif // MATHCORE_IGENERALFUNCTIONWITHGRADIENT_H


// File IParameterizedFunctionWithGradient.h
#ifndef MATHCORE_IPARAMETERIZEDFUNCTIONWITHGRADIENT_H
#define MATHCORE_IPARAMETERIZEDFUNCTIONWITHGRADIENT_H

#include "IParameterizedFunction.h"
#include "IGeneralFunctionWithGradient.h"
#include <vector>

typedef std::vector<double> dblvec; // exposition only

class IParameterizedFunctionWithGradient // IPFwG
: public IParameterizedFunction
, public IGeneralFunctionWithGradient
{
    // Interface for single-argument functions that also supply a gradient
    // and that have bindable parameters,

public:
    virtual ~IParameterizedFunctionWithGradient() { /**/ }

    virtual IParameterizedFunctionWithGradient * clone() const = 0;

    void parametersGradient( double x, dblvec & g ) const {
        do_parametersGradient(x, g);
    }

private:
    virtual double do_evaluateFunction( double ) const = 0;
    virtual void do_changeParameters( dblvec const & ) = 0;
    virtual size_t do_numberOfParameters() const = 0;
    virtual void do_fetchParameters( dblvec & ) const = 0;
    virtual double do_gradient( double ) const = 0;
    virtual void do_parametersGradient( double, dblvec & ) const = 0;

}; // IPFwG

#endif // MATHCORE_IPARAMETERIZEDFUNCTIONWITHGRADIENT_H

```

These five interfaces describe *how* "general functions" can be used, but none is a concrete class. The current CppFunc contains two concrete classes as parts of this hierarchy: CParamFunction and Polynomial.

It seems the purpose of CParamFunction is to wrap a pointer-to-function (of the appropriate signature) in the interface provided by IParamFunction. We believe this functionality can be provided and extended under our proposed redesign. The extended flexibility includes:

- the ability to wrap *any* callable object, not just free functions, and
- the ability to wrap objects with a wider variety of signatures.

We believe this can be achieved using a small set of class templates. The first, WrappedFunction, can wrap any callable object with the correct signature: one argument, of type double. It implements the IGeneralFunction interface.

```
// File WrappedFunction.h
#ifndef MATHCORE_WRAPPED_FUNCTION_H
#define MATHCORE_WRAPPED_FUNCTION_H

#include "IGeneralFunction.h"

template< class CALLABLE >
class WrappedFunction
    : public IGeneralFunction
{
    // Concrete class, wraps a callable object which takes one
    // argument, and provides the IGeneralFunction interface.

public:
    explicit WrappedFunction( CALLABLE f ) : m_f( f ) { /**/ }

    virtual WrappedFunction * clone() const {
        return new WrappedFunction( *this );
    }

    virtual ~WrappedFunction() { /**/ }

private:
    CALLABLE m_f;

    virtual double do_evaluateFunction( double x ) const {
        return m_f( x );
    }
}; // WrappedFunction

#endif // MATHCORE_WRAPPED_FUNCTION_H
```

The second, WrappedParameterizedFunction, can wrap any callable object with the correct signature: two arguments, the first a double, the second a `std::vector<double> const&`. This is similar to the original CParamFunction, but does not require the callable object to be a free function.

```
// File WrappedParameterizedFunction.h
#ifndef MATHCORE_WRAPPEDPARAMETERIZEDFUNCTION_H
```

```
#define MATHCORE_WRAPPEDPARAMETERIZEDFUNCTION_H

#include "IPparameterizedFunction.h"
#include <vector>

typedef std::vector<double> dblvec; // exposition only

template <class CALLABLE>
class WrappedParameterizedFunction
: public IPparameterizedFunction
{
    // Concrete class, wraps a callable object which takes two
    // arguments, and provides the IPparameterizedFunction interface.

public:
    WrappedParameterizedFunction( CALLABLE f, dblvec const & p ) :
        m_f( f ),
        m_parameters( p )
    { /**/ }

    virtual WrappedParameterizedFunction * clone() const {
        return new WrappedParameterizedFunction( *this );
    }

    virtual ~WrappedParameterizedFunction() { /**/ }

private:
    CALLABLE m_f;
    dblvec m_parameters;

    virtual double do_evaluateFunction( double x ) const {
        return m_f( x, m_parameters );
    }

    virtual void do_changeParameters( dblvec const & newparams ) {
        if( newparams.size() != numberOfParameters() )
            throw /* appropriate error */;
        m_parameters = newparams;
    }

    virtual size_t do_numberOfParameters() const {
        return m_parameters.size();
    }

    virtual void do_fetchParameters( dblvec & p ) const {
        p = m_parameters;
    }

}; // WrappedParameterizedFunction

#endif // MATHCORE_WRAPPEDPARAMETERIZEDFUNCTION_H
```

Note that we do *not* suggest providing additional class templates to deal with "functions that also supply gradients". This is, in part, because we expect pre-existing objects of this type to be uncommon. Furthermore, we would suggest that users who need the calculation of gradients rely on the `Deriv` package. We believe this can be done with no loss of efficiency and with a significant gain in generality as described in our analysis of `Deriv` (see Differentiation, below).

5.2.5 Chebyshev

The class `Chebyshev`, while part of the `CppFunc` module, is not a member of the `IGenFunction` hierarchy. Rather, it is a *user* of the hierarchy.

Some of the overloaded functions (e.g., `evalErr`) are separated in the header, making it harder to notice that they form an overload set.

`Chebyshev` seems partly designed for polymorphic use:

- it has a virtual destructor, and
- it has a `protected` function.

Other than destruction, `Chebyshev` appears to lack any potentially polymorphic functionality, however. Why would one inherit from `Chebyshev`? Unless there is a reason we have missed, it seems that the virtual destructor and `protected` function are not needed, and that the class is really *not* suitable for use as a base class. If there *is* a case for polymorphic use, then the functions which are intended to be overridden in derived classes must be identified, and declared `virtual`.

The copy constructor and copy assignment operator are declared `private`, with a note that "usually copying is non trivial." Nonetheless each function *is* implemented, and the implementations are unusual. It would seem better, if copying is to be forbidden, to leave the declaration `private` and to provide *no* implementation -- thus avoiding possibly erroneous use of these functions by other member functions.

It is not clear *why* copying is considered non-trivial. It seems that the meaning of copying a `Chebyshev` object is clear. What may be non-trivial is correct *implementation* of such copying. It seems that insufficient attention has been paid to the design of the class's member objects, and to their management. The contained bare pointers would be better replaced by pointer-like classes (see *Resource management_* above) which automate the memory management.

6 Brief Analysis of Selected Packages

In this section, we consider (more briefly) two of the packages that use the "core functionality" provided by `CppFunc`. We have selected `Integration` and `Deriv` because they exemplify the use of the interfaces provided by `CppFunc`.

6.1 Integration

6.1.1 IIntegrator

The `Integration` modules consist of a single concrete class, `GSLIntegrator`, which is also known through a typedef as `Integrator`. This class is the sole class derived from the interface `IIntegrator`.

It seems that the addition of this base class is premature. Since there is only a single class in the inheritance hierarchy, there is nothing for current users to gain from use of the base interface. Furthermore, useful functionality of the `GSLIntegrator` class (such as the ability to set the integration rule via `setIntegrationRule`) is not available through the base class. Until there are several more concrete classes that perform integration available, it seems best to reduce the complexity of the library by removing the `IIntegrator` interface. There seems to be no other part of **MATHLIB** that depends on this interface.

6.1.2 GSLIntegrator

The class `GSLIntegrator` (also known through a typedef as `Integrator`) has a two-part interface:

- one part is the interface inherited from `IIntegrator`, and
- the other part is expressed via member templates (which can not be virtual).

As noted above, we suggest removal of the base class `IIntegrator`.

We believe that the template members of `GSLIntegrator` are not needed. They seem to be present in order to make it convenient to "wrap" a simple function. Instead, the wrappers suggested above (see A proposed refactoring) provide this functionality. Relying upon those wrappers' functionality allows `GSLIntegrator` to concentrate on the task of *integration*, making it significantly simpler and thus easier to maintain or extend. We suggest the template members of `GSLIntegrator` be removed, and that `GSLIntegrator` rely upon the interface `IGeneralFunction`.

We believe the following functions are sufficient to provide for filling the `gsl_function` struct which is needed to call the C-language GSL integration routines used by `GSLIntegrator`.

```
// GSLstuff.cc

#include "IGeneralFunction.h"
#include "gsl/gsl_math.h"

extern "C"
double applyIGF( double x, void * p )
{
    IGeneralFunction * igf_p = static_cast<IGeneralFunction *>( p );
    return (*igf_p)( x );
}

void fillGSLFunction( IGeneralFunction const & igf, gsl_function & gsl_f )
{
    gsl_f.function = & applyIGF;
    gsl_f.params
        = static_cast<void *>( const_cast<IGeneralFunction *>( & igf ) );
}
```

We note that this class has three constructors, each with default values provided for each constructor parameter:

- Since each such constructor can be invoked with a single argument, it can be used to perform implicit conversions. This behavior would, at best, surprise the unwary user, and so the noted constructors should be declared `explicit`.
- Overload resolution will produce an ambiguity for such an overload set when default-construction is attempted, and even when single-argument (of enum type) construction is attempted.

Futhermore, the utility of any default-constructed `GSLIntegrator` object is unclear to us.

6.2 Differentiation

Time did not permit us to investigate the module `Deriv` at the same level we investigated `Integration`. However, we wish to make the same point: `Deriv` should rely on the functionality provided by the interfaces in `CppFunc`. Wrappers for user functions should be provided within `CppFunc`, not within `Deriv`.

Because the CppFunc module presents interfaces which capture the concept of differentiation, it is possible to make the class which performs differentiation (1) employ (invoke) this functionality when present, and (2) perform the work of numerical differentiation when it is not present. This allows users to rely on one class to perform differentiation, and to obtain the efficiency of the analytic calculation from those functions which provide it.

We believe the following sketch shows the key part of the implementation, making use of the IGradient interface from above.

```
#include "IGeneralFunction.h"
#include "IGradient.h"

class IDifferentiationWorker // etc.

class GSLDifferentiator
: public IDifferentiationWorker // etc.

class ExactDifferentiator
: public IDifferentiationWorker // etc. -- uses gradient()

class Differentiator
{
public:
    Differentiator( IGeneralFunction const & igf )
        : m_d_worker( 0 == dynamic_cast<IGradient>( igf )
                    ? new GSLDifferentiator ( igf )
                    : new ExactDifferentiator( igf )
                  )
    { /**/ }

    // other functions ...

private:
    IDifferentiationWorker * m_d_worker;
}; // Differentiator
```

The class Differentiator (rather than Derivator) is the class users use to perform differentiation. It depends on an interface class, IDifferentiationWorker, which presents the interface for numerical differentiation. GSLDifferentiationWorker implements this interface, and uses the gsl library to perform the numeric differentiation. (It is like the current GSLDerivator.) The class ExactDifferentiator also implements this interface; this class contains a pointer to an object having the IGradient interface, and calls upon this object to evaluate the derivative (ignoring the unneeded step size).

7 Conclusion

We believe that version 0.0.3 of **MATHLIB** is a significant step forward in the design and implementation of the library. With modifications of the sort we have proposed, we believe the library will provide a solid basis for the sorts of mathematical manipulations that will be needed for the work of CMS.