Implementation By Implication

Shayne Fletcher

Abstract

In this article, we demonstrate the application of various modern C++ implementation techniques in the development of a computer program to solve the classic problem of computing Black & Scholes implied option volatilities.

***Keywords*—** implied option volatility, C++, generic programming, policy based design, template metaprogramming, named template parameters

1 Introduction

In 2006, Peter Jaeckel in a very clever paper, "By Implication" [Jaeckel, 2006], examines the issues with computing the Black & Scholes implied option volatility given an observable market price. Since that paper explains the issues surrounding and suggested solutions for robust and efficient computation of volatility numbers much more elegantly than we could hope to do here, we don't intend to cover those details. Rather, we will take the algorithm outlined there mathematically and use it as an opportunity to employ some techniques of modern C++, namely, generic programming, policy-based design and named template parameters to implement Peter's ideas as a flexible library component.

2 The Basic Idea

More or less quoting verbatim from [Jaeckel, 2006], a plain vanilla call or put option price p in the Black-Scholes-Merton [Black and Scholes, 1973, Merton, 1973] framework is given by

$$p = \theta \cdot \left[F \cdot \Phi\left(\theta \cdot \left[\frac{\ln(F/K)}{\sigma} + \frac{\sigma}{2}\right]\right) - K \cdot \Phi\left(\theta \cdot \left[\frac{\ln(F/K)}{\sigma} - \frac{\sigma}{2}\right]\right)\right]$$

where $\theta = 1$ for call options and $\theta = -1$ for put options, $F := Se^{(r-d)T}$, S is the current spot, K is the strike, r is a flat continuously compounded interest rate to maturity, d is a flat continuously compounded dividend rate, $\sigma = \hat{\sigma} \cdot \sqrt{T}$, $\hat{\sigma}$ is the root-mean-square lognormal volatility, T is the time to expiry and $\Phi(\cdot)$ is the standard cumulative normal distribution function. Jaeckel's method is an efficient and robust algorithm for inverting the above equation to find the implied volatility $\hat{\sigma}$ given parameters p, F, K, θ and T.

Now for the reader for which the above is of little or no familiarity ... fear not! This is a note about modern C++ techniques. A deep or even passing understanding of this algorithm in theory or code is not required to get the benefits of the discussions to follow.

Deferring for the time being any algorithmic details and working simply from the description provided so far, the most basic starting point for our program we'll phrase as the the following class design.

```
class jaeckel_method
{
public:
  enum status_t
  {
    undetermined, determined, converged, max_iterations_exceeded
  };

public:
  jaeckel_method
  (
      double p
    , double F
    , double K
    , double T
    , e_call_put theta
    , unsigned long max_its //max number of iterations permitted
    , double eps  //tolerance for relative convergence
  );
  double as_number() const { return vol_; }
  int status() const { return status_; }
  unsigned long iterations() const { return iterations_; }

private:
  int status_;
  double vol_;
  unsigned long iterations_;
};
```

The idea of this design is that the algorithm is applied in the constructor and thereafter the object constructed can be queried for the result. Any errors resulting during application of the algorithm or reasons detected that would prevent the algorithm from being applicable we'll assume for the time being as signalled by an exception thrown from the constructor.

3 Generic Programming

On to the first generalization[1]. The design of the preceding section fixed the representation of floating point numbers strictly to the native `double` precision type. Now this hard decision is somewhat unfortunate and would preclude the library component from being used with higher precision floating points such as `long double` for example, or an arbitrary precision floating point number type such as the `RR` type provided the "NTL" library [2]. We'll address this deficiency by making `jaeckel_method` a class template in the floating point type.

```
template<class RealT=double>
class jaeckel_method
{
public:
  // ...

public:
  jaeckel_method
  (
      RealT p
    , RealT F
    , RealT K
    , RealT T
    , e_call_put theta
    , unsigned long max_its
    , RealT eps
  );

  RealT as_number() const { return vol_; }

  //...
};
```

In the above, we declare our intention that the algorithm will work with any type *RealT* that meets the conceptual requirements of a Boost.Math_Toolkit [3] *RealType*. In short, those conceptual requirements basically amount to saying that a *RealT* "behaves" just like floating built-in types.

Before we leave this section, look again to the last two parameters to the class constructor. In fact, those numbers, the maximum number of iterations to permit and the convergence tolerance are more suitably determined by calculations based on the precision of the floating point type than left to the library user to provide. A simple way to handle that is to change the constructor such that it becomes a template in two function objects, one to compute the maximum number of iterations, another to compute the tolerance.

```
template <class ItsF, class EpsF>
jaeckel_method(
    RealT p
  , RealT F
  , RealT K
  , RealT T
  , e_call_put theta
  , ItsF max_its_f
  , EpsF eps_f
);
```

These function object types are expected to satisfy a *NullaryFunction* concept, in that they take no arguments. The library can then offer suitable default definitions such as, for example, the following:

[1] For more information about generic programming than the little presented here, see http://www.generic-programming.org/

[2] NTL:A portable C++ library for doing Number Theory. See http://shoup.net/ntl/ for further details.

[3] Boost is a collection of peer-reviewed portable C++ libraries. See boost.org for further details.

```
template <class RealT=double>
struct jaeckel_method_default_iterations
{
  unsigned long operator()() const
  {
    using std::abs;
    using std::log;

    RealT eps=boost::math::tools::epsilon<RealT>();

    return boost::numeric_cast<unsigned long>
                    (abs(log(eps)/log(RealT(2))));
  }
};

template <class RealT=double>
struct jaeckel_method_default_tolerance
{
  RealT operator()() const
  {
    using std::sqrt;

    RealT eps=boost::math::tools::epsilon<RealT>();
    RealT sqrt_eps=sqrt(eps);

    return sqrt_eps*sqrt(sqrt_eps); //eps^(3/4)
  }
};
```

In the event either one or both of the library provided defaults don't meet a specific library user's needs, they can simply write their own definitions for the case at hand and supply those to the algorithm instead, and that, is the essence of policy-based design, the topic of the next section.

4 Policy-Based Design

The numerical aspects of Jaeckel's method for computing implied volatility can be coded very elegantly and concisely. Nonetheless, in the real world, this program can go horribly wrong for a variety of reasons:

- a negative forward rate is given;
- a negative strike is given;
- a negative time to maturity is given;
- a put option price is greater than the strike;
- a call option price is greater than the forward rate;
- the option price provided is less than the option's intrinsic value;
- for whatever reason, a non-finite volatility is detected during application of the algorithm.

Of course, each of these error conditions is detectable and since the C++ language provides programmers with the ability to throw exceptions, the library developer implementing Jaeckel's method, is likely to do so on detecting one of these error conditions.

The thing is, computing implied volatility occurs so often in finance that in some such contexts throwing exceptions on failure might be the right thing to do but in other contexts not! For some applications, it might be more appropriate to continue execution and simply return a prespecified volatility. If the library were to "bake" in the exception throwing error handling behavior into the library component, a later client with different error handling requirements would at best be hindered and at worst be unable to use the component (likely leading to code replication) and that would be a shame.

Enter policy-based design: a better solution for the library component is to provide sensible default error handling behavior but allow the user (library client) to customise it for their contexts if and when they need to. One thing though, note that the algorithm has been carefully crafted such that if nonsensical inputs have been correctly detected, no non-finite volatility can be achieved and if one were to be encountered it can therefore only be the result of a programming error and so better handled by an `assert`.

Taking the idea of the overridable error handlers into account suggests a class design like the following:

```
template
<
    class RealT = double
  , class E1 = default_e1 //negative fwd
  , class E2 = default_e2 //negative strike
  , class E3 = default_e3 //negative time to maturity
  , class E4 = default_e4 //put price > strike
  , class E5 = default_e5 //call price > fwd
  , class E6 = default_e6 //price < intrinsic
>
class jaeckel_method
{
//...
};
```

The idea now is that error handling policy types for each of the overridable error conditions are given, the library providing defaults. When an error is detected in the algorithm, a static function of the appropriate policy type is invoked to handle the error. For example, the (exception throwing) default handler for the negative forward error might read:

```
//library provided default handler for negative forward
struct default_e1
{
   typedef boost::fusion::vector<int, double, unsigned long> return_type;

   static return_type on_negative_forward (
     double p,double F,double K, double T, e_call_put theta
   , unsigned long its, double eps)
   {
     throw std::runtime_error("negative forward");

     return return_type();
   }
};
```

If the library provided default error handling policies are acceptable, an instance of the algorithm could be instantiated with the syntax `jaeckel_method<>`. The wrinkle now though is that if a non-default argument must be specified, all preceding arguments must be specified too (even though they may have their default values). What would be better again would be to provide an interface whereby the library client is able to just override what they need, say with a syntax like (not C++!) `jaeckel_method<negative_strike=my_custom_policy>`. What we will work on from here is to organize things so that we end up as close to that interface as we can get. Specifically, we will arrive at a syntax that would read `jaeckel_method<negative_strike<my_custom_policy> >` for the case above. If two (or more) policies were to be overriden we would write

```
    jaeckel_method<
        negative_strike<my_custom_policy>
      , put_price_greater_than_strike<another_custom_policy> >
```

for example (of course we will allow for the 'named' arguments in *any* order so that the type

```
    jaeckel_method<
         put_price_greater_than_strike<another_custom_policy>
      ,  negative_strike<my_custom_policy> >
```

would be equivalent to the preceding example).

5 Named Template Parameters

The basic technique we are about to apply is detailed in [Vandervoorde and Josuttis, 2003] and we will follow that exposition fairly closely (but also extend beyond what's presented there and wrap up what we can as library components for building solutions of this kind). The key idea is to place default policy types in a base class and override some of them through inheritance. Rather than directly supplying the type arguments we provide them indirectly through helper classes. For example, as seen in the last section, we might write `jaeckel_method<negative_strike<my_custom_policy> >`. In that construct, for all but the negative strike error the default policies are to apply but in the case of the negative strike error, the user provided `my_custom_policy` is to override the default. Given the above it is seen that any template argument can describe any of the policies and so it follows that the defaults cannot be different. In terms of the class at hand, this means the following:

```
struct jaeckel_method_err_defaults;

template
<
   class RealT=double
 , class E1 = detail::jaeckel_method_err_defaults //negative fwd.
 , class E2 = detail::jaeckel_method_err_defaults //negative strike
           .
           .
           .
 , class E6 = detail::jaeckel_method_err_defaults //price < intrinsic
>
class jaeckel_method
{
private:
  typedef boost::mpl::vector<E1,E2,E3,E4,E5,E6> policies_t;
  typedef meta::policy_selector<policies_t> err_hnds_t;
  typedef typename err_hnds_t::err_hnd1_t err_hnd1_t; //negative fwd.
  typedef typename err_hnds_t::err_hnd2_t err_hnd2_t; //negative strike
           .
           .
           .
  typedef typename err_hnds_t::err_hnd6_t err_hnd6_t; //price < intrinsic

public:

 //...

};
```

Look now to the definition of `policies_t` in the code above. It is a typedef for a `boost::mpl::vector<E1,...,E6>`. What is an MPL vector? It is a type list. That is a C++ type representing a collection of types. Type lists were popularized by Andrei Alexandrescu in his book "Modern C++ Design" [Alexandrescu, 2001]. The Boost.MPL library (MPL is an acronym for "Metaprogramming library") is a library of types and tools for template metaprogramming tasks and provides several implementations of type-lists and algorithms to operate on them. Abrahams and Gurtovoy in their book "C++ Template Metaprogramming" [Abrahams and Gurtovoy, 2001] provide an excellent source of information on template metaprogramming, its applications and use of the Boost.MPL library to facilitate it. Now note the definition of `err_hnds_t` as a typedef for `meta::policy_selector<T>` where `T` is the list of policies just described. This is a type that merges all the template arguments into a single type overriding default error policies with any non-default policies provided. As mentioned earlier this will be achieved by inheritance and the development of this type is what we will be turning our attention to now.

5.1 Policy Selector

We start with the following simple definitions:

```
namespace meta
{
  template <class BaseT, int D>
  struct discriminator : BaseT {};

  template <class T, class D>
  struct generate_discriminator { typedef discriminator<T, D::value> type; };

}//namespace meta
```

The purpose of `template<class, int> class discriminator` is to allow for inheritance from the same base class more than once (in C++ it is not legal to have multiple direct base classes of the same type). The template class `template <class,class> generate_discriminator` is a *metafunction*, i.e. a type that embodies a compile time computation much like a function represents a runtime computation. For example, the declaration might be interpreted as a template class that given "arguments" `T` and `D` computes the type `discriminator<T, D::value>` and "returns" the result in its `type` member typedef. What we are edging towards is, given a typelist with elements `T1,..,Tn` say, to produce a list of types `discriminator<T1, 1>,...,discriminator<Tn, n>`.

The next step is to generate from `L=T1,..Tn` the list of integers $1..n$. That list we will model as a `boost::mpl::vector_c<int>`, that is a type list the elements of which are `boost::mpl::int_c<i>` types where $i = 1 \ldots n$. At the heart of the procedure, are the following *metafunction classes* (a metafunction class is a class with a publicly accessible nested metafunction called `apply`):

```
namespace meta
{
  namespace detail
  {
    template <bool=true>
    struct dimension
    {
      template <class SeqT>
      struct apply
      {
        typedef boost::mpl::int_<1>::type type;
      };
    };

    template <>
    struct dimension<false>
    {
     template <class SeqT>
     struct apply
      {
        typedef typename
          boost::mpl::next
          <
            typename boost::mpl::back<SeqT>::type
          >::type type;
      };
    };

  }//namespace detail
}//namespace meta
```

The idea here is that the template parameter `SeqT` is a `mpl::vector_c<int>` type as described above. One of the above metafunction classes will be selected depending upon the emptiness of `SeqT`. If `SeqT` is empty, the first specialization of `dimension` will be selected. The resulting type computed will be `boost::mpl::int_<1>`. If `SeqT` is not empty, the type at the back of the list (`boost::mpl::int_<i>` say) will be inspected and a new type obtained by application of the `boost::mpl::next` metafunction (producing `boost::mpl::int_<i+1>` say). Now the code to apply the metafunctions above to produce the list of integers:

```
namespace meta
{
  namespace detail
  {
    struct add_dimension
    {
      template <class T, class>
      struct apply
      {
        typedef typename
          boost::mpl::push_back
          <
             T
           , typename
               boost::mpl::apply
               <
                   dimension<boost::mpl::empty<T>::value>
                 , T
               >::type
          >::type type;
      };
    };

    struct generate_dimensions
    {
      template <class SeqT>
      struct apply
      {
        typedef typename
          boost::mpl::accumulate
          <
              SeqT
            , boost::mpl::vector_c<int>::type
            , add_dimension
          >::type type;
      };
    };

  }//namespace detail
}//namespace meta
```

Let's start with the metafunction class `generate_dimensions`. The template parameter to its nested metafunction `apply` is, in this case, the incoming list of policy types (the error handler policies). The type computed by `apply` is the `boost::mpl::vector_c<int>` containing types representing $1..n$ in order. This is achieved by invocation of the `boost::mpl::accumulate` metafunction (a compile time analogue of the STL's `accumulate`), the accumulating "operation" being embodied by the `add_dimension` metafunction class. The `accumulate` metafunction reaches into the `add_dimension` metaclass for its `apply` metafunction passing through the current list of integers. The effect of this apply is to produce a new list with the "next" element appended to the back by application of the `dimension` metafunctions examined earlier. Here is a (compile time) test that demonstrates that all that machinery works as planned:

```
//strictly compile time test

void generate_dimensions_test()
{
  typedef
    boost::mpl::apply
    <
        meta::detail::generate_dimensions
      , boost::mpl::vector<char, int, long, double>
    >::type dimensions_type;

  typedef
    boost::mpl::vector4
    <
        boost::mpl::int_<1>
      , boost::mpl::int_<2>
      , boost::mpl::int_<3>
      , boost::mpl::int_<4>
     > expected_type;

  BOOST_MPL_ASSERT((
    boost::mpl::equal<dimensions_type, expected_type> ));
}
```

So we now have two lists : `L=T1...Tn` are the policy types, and `N=1..n` the list of integers, one for each policy in order. What we will want to do is apply a metafunction to pairs from these two lists in order to produce the list `discriminator<T1,1>..discrimanator<Tn, n>`. Finally, we will want to produce a class that inherits from each of the elements of that list. We achieve that with a metafunction we call `axes`:

```
namespace meta
{
  template <class List, class F>
  struct axes
  {
    typedef typename
      boost::mpl::transform
      <
          List
        , typename
            boost::mpl::apply
            <
               detail::generate_dimensions
             , List
            >::type
        , F
      >::type axis_types;
```

```
    typedef typename
      boost::mpl::inherit_linearly
      <
          axis_types
        , boost::mpl::inherit
          <
             boost::mpl::_1
           , boost::mpl::_2
          >
      >::type type;

    enum { dimensions =
      boost::mpl::size<axis_types>::type::value };
  };

}//namespace meta
```

As can be seen from the above, the list `F<T1, 1>,...,F<Tn, n>` is produced by the `boost::mpl::transform` metafunction (analogue to the STL's runtime `transform` function) by application of an as yet unspecified metafunction `F`. The application of `boost::mpl::inherit_linearly` completes the job of producing a class inheriting from all of the types in the `F<T1,1>,...,F<Tn,n>` list. One thing might require explaining at this point. That is, the presence of `boost::mpl::_1` and `boost::mpl::_2` in the above (the second argument to the application of `inherit_linearly`). Well, they are so-called MPL *placeholder* types and the expression

```
boost::mpl::inherit
<
  boost::mpl::_1
, boost::mpl::_2
>
```

is termed a placeholder expression. The point is that `boost::mpl::inherit` is a template, not a type. Introducing the placeholders makes a type from the template and so can be passed as an argument to `boost::mpl::inherit_linearly`. The Boost metaprogramming library has smarts built in to deal with such placeholder expressions in addition to metafunction classes.

Now, in the above, what will `F` be? We met it earlier. It is the metafunction `generate_discriminator`. Finally, writing `policy_selector` is simple:

```
namespace meta
{
  template <class Policies>
  struct policy_selector
    : axes
      <
          Policies
        , generate_discriminator
          <
             boost::mpl::_1
           , boost::mpl::_2
          >
    >::type
  {};

}//namespace meta
```

Here is another compile time test that demonstrates what we have achieved:

```
struct policy1 {};
struct policy2 {};

void policy_selector_test()
{
  typedef boost::mpl::vector<policy1, policy2> policies;
  typedef meta::policy_selector<policies> selector;

  BOOST_MPL_ASSERT((
    boost::is_base_of<
      meta::discriminator<policy1, 1>, selector> ));
  BOOST_MPL_ASSERT((
    boost::is_base_of<
       meta::discriminator<policy2, 2>, selector> ));
}
```

The above tests the `selector` type has both of
`meta::discriminator<policy1, 1>` and
`meta::discriminator<policy2, 2>` as base types.

5.2 Finishing Off

With `class policy_selector` at our disposal, the job at hand completing the customizable error handling framework for Jaeckel's method, is readily finished.

First we write a class for the default error handling (for the sake of brevity we present the complete implementation of the default error behavior for two cases, only the error messages changing in the implementation of the others):

```
namespace result_of
{
  template <class RealT>
  struct jaeckel_method_err
  {
    typedef boost::fusion::vector<int, RealT, unsigned long> type;
  };

}//namespace result_of

struct jaeckel_method_default_err_handler_impl
{
  template<class RealT>
  static typename result_of::jaeckel_method_err<RealT>::type
  on_negative_forward(
      RealT p, RealT f, RealT K, RealT T, e_call_put theta
    , unsigned long its, RealT eps)
  {
    typedef typename
      result_of::jaeckel_method_err<RealT>::type return_type;

    throw std::runtime_error("negative forward");

    return return_type();
  }

  //on_negative_strike, on_negative_time_to_maturity,
  //on_put_price_less_than_strike, on_call_price_greater_than_forward...

  template<class RealT>
  static typename result_of::jaeckel_method_err<RealT>::type
  on_price_less_than_intrinsic(
      RealT p, RealT f, RealT K, RealT T, e_call_put theta
    , unsigned long its, RealT eps)
  {
    typedef typename
      result_of::jaeckel_method_err<RealT>::type return_type;

    throw std::runtime_error("price less than intrinsic value");

    return return_type();
  }
};
```

We wrap that up in a class that also exports default error handler typedefs:

```
struct jaeckel_method_default_err_handlers
  : jaeckel_method_default_err_handler_impl
{
  typedef jaeckel_method_default_err_handler_impl base_t;

  typedef base_t err_hnd1_t; //negative fwd.
  typedef base_t err_hnd2_t; //negative strike
            .
            .
            .
  typedef base_t err_hnd6_t; //price < intrinsic
};
```

We need to be careful to avoid ambiguity if we end up inheriting multiple times from this base class and so write (note the use of virtual inheritance):

```
struct jaeckel_method_err_defaults
  : virtual jaeckel_method_default_err_handlers
{};
```

Lastly we provide the helper classes for overriding the default typedefs:

```
template <class P>
struct negative_forward
  : virtual detail::jaeckel_method_default_err_handlers
{
  typedef P err_hnd1_t;
};

template <class P>
struct negative_strike
  : virtual detail::jaeckel_method_default_err_handlers
{
  typedef P err_hnd2_t;
};

//negative_time_to_maturity, price_greater_than_strike,
//call_price_greater_than_forward...

template <class P>
struct price_less_than_intrinsic
  : virtual detail::jaeckel_method_default_err_handlers
{
  typedef P err_hnd6_t;
};
```

Here is a snippet of code that shows the error handling customization in action:

```
struct my_err_handler
{
  template <class RealT>
  static boost::fusion::vector<int, RealT, unsigned long>
  on_put_price_greater_than_strike(
    RealT,RealT,RealT,RealT,e_call_put,unsigned long,RealT)
  {
```

```
    typedef
      boost::fusion::vector<int, RealT, unsigned long> return_type;

    return return_type(2, (std::numeric_limits<RealT>::max)(), 0ul);
  }
};

void test_jaeckel_method()
{
  typedef
    jaeckel_method<
      double
    , put_price_greater_than_strike<my_err_handler > > imp_vol_t;

  jaeckel_method_default_tolerance<> eps;
  jaeckel_method_default_iterations<> its;

  double f, k, p, t;

  // negative forward
  f = -0.05;
  k =  0.07;
  p =   1.0;
  t =   1.0;
  BOOST_CHECK_THROW(
    (imp_vol_t(p, f, k, t, call, its, eps)), std::runtime_error);

  // put price greater than strike
  f =  0.07;
  k =  0.05;
  p =  0.08;
  t =   1.0;
  imp_vol_t vol2(p, f, k, t, put, its, eps);
  BOOST_CHECK_EQUAL(vol2.status(), imp_vol_t::determined);
  BOOST_CHECK_EQUAL(vol2.as_number(), (std::numeric_limits<double>::max)());
  BOOST_CHECK_EQUAL(vol2.iterations(), 0);
}
```

Lastly in full detail, here is the complete implementation of Jaeckel's method with customizable error handling via named template parameters.

```
#if !defined(JAECKEL_METHOD_DDAE8974_C6E8_40B3_AD3A_43417A3B1CAC_INCLUDED)
# define JAECKEL_METHOD_DDAE8974_C6E8_40B3_AD3A_43417A3B1CAC_INCLUDED

# if defined(_MSC_VER) && (_MSC_VER >= 1020)
#   pragma once
# endif// defined(_MSC_VER) && (_MSC_VER >= 1020)

# include <cppf/maths/config.hpp>
# include <cppf/maths/norm_cdf.hpp>
# include <cppf/maths/inverse_norm_cdf.hpp>
# include <cppf/maths/e_call_put.hpp>
# include <cppf/maths/heaviside.hpp>
# include <cppf/meta/policy_selector.hpp>

# include <boost/fusion/container/vector.hpp>
# include <boost/fusion/container/generation/make_vector.hpp>
# include <boost/fusion/container/generation/vector_tie.hpp>
# include <boost/fusion/tuple/tuple_tie.hpp>
# include <boost/numeric/conversion/cast.hpp>

# include <limits>
# include <cmath>
# include <stdexcept>
# include <cassert>

namespace cppf { namespace maths { namespace process { namespace lognormal {

namespace implied_vol
{
  namespace detail
  {
    struct jaeckel_method_err_defaults;

  } // namespace detail

  template <class RealT=double>
  struct jaeckel_method_default_iterations
  {
    unsigned long operator()() const
    {
      using std::abs;
      using std::log;

      RealT eps=boost::math::tools::epsilon<RealT>();

      return boost::numeric_cast<unsigned long>(abs(log(eps)/log(RealT(2))));
    }
  };

  template <class RealT=double>
  struct jaeckel_method_default_tolerance
  {
    RealT operator()() const
    {
      using std::sqrt;

      RealT eps=boost::math::tools::epsilon<RealT>();

      RealT sqrt_eps=sqrt(eps);

      return sqrt_eps*sqrt(sqrt_eps); //eps^(3/4)
    }
  };

  template
  <
     class RealT=double
   , class E1 = detail::jaeckel_method_err_defaults //negative fwd.
   , class E2 = detail::jaeckel_method_err_defaults //negative strike
   , class E3 = detail::jaeckel_method_err_defaults //negative time
   , class E4 = detail::jaeckel_method_err_defaults //put price > strike
   , class E5 = detail::jaeckel_method_err_defaults //call price > fwd.
   , class E6 = detail::jaeckel_method_err_defaults //price < intrinsic
  >
  class jaeckel_method
  {
public:
    enum status_t
    {
        undetermined=1
      , determined
      , converged
      , max_iterations_exceeded
    };

public:
    template <class ItsF, class EpsF>
    jaeckel_method
    (
        RealT price
      , RealT forward
      , RealT strike
      , RealT time_to_maturity
      , e_call_put call_put_code
      , ItsF max_its_f
      , EpsF eps
    );
    RealT as_number() const { return vol_; }
    int status() const { return status_; }
    unsigned long iterations() const { return iterations_; }

private:
    int status_;
    RealT vol_;
    unsigned long iterations_;
  };

  namespace detail
  {
    namespace result_of
    {
      template <class RealT>
      struct jaeckel_method_err
      {
        typedef
          boost::fusion::vector
          <
              int
            , RealT
            , unsigned long
          > type;
      };

    }//namespace result_of

    struct jaeckel_method_default_err_handler_impl
    {
      template<class RealT>
      static typename result_of::jaeckel_method_err<RealT>::type
      on_negative_forward(
          RealT price
        , RealT fwd
        , RealT strike
        , RealT t
        , e_call_put cp
        , unsigned long its
        , RealT eps)
      {
        typedef typename
          result_of::jaeckel_method_err<RealT>::type return_type;

        throw std::runtime_error("negative forward");

        return return_type();
      }

      template<class RealT>
      static typename result_of::jaeckel_method_err<RealT>::type
      on_negative_strike(
          RealT price
        , RealT fwd
        , RealT strike
        , RealT t
        , e_call_put cp
        , unsigned long its
        , RealT eps
      )
```

```
  {
    typedef typename
      result_of::jaeckel_method_err<RealT>::type return_type;

    throw std::runtime_error("negative strike");

    return return_type();
  }

  template<class RealT>
  static typename result_of::jaeckel_method_err<RealT>::type
  on_negative_time_to_maturity(
      RealT price
    , RealT fwd
    , RealT strike
    , RealT t
    , e_call_put cp
    , unsigned long its
    , RealT eps
  )
  {
    typedef typename
      result_of::jaeckel_method_err<RealT>::type return_type;

    throw std::runtime_error("negative time to maturity");

    return return_type();
  }

  template<class RealT>
  static typename result_of::jaeckel_method_err<RealT>::type
  on_put_price_greater_than_strike(
      RealT price
    , RealT fwd
    , RealT strike
    , RealT t
    , e_call_put cp
    , unsigned long its
    , RealT eps
  )
  {
    typedef typename
      result_of::jaeckel_method_err<RealT>::type return_type;

    throw std::runtime_error("put price greater than strike");

    return return_type();
  }

  template<class RealT>
  static typename result_of::jaeckel_method_err<RealT>::type
  on_call_price_greater_than_forward(
      RealT price
    , RealT fwd
    , RealT strike
    , RealT t
    , e_call_put cp
    , unsigned long its
    , RealT eps
  )
  {
    typedef typename
      result_of::jaeckel_method_err<RealT>::type return_type;

    throw std::runtime_error("call price greater than forward");

    return return_type();
  }

  template<class RealT>
  static typename result_of::jaeckel_method_err<RealT>::type
  on_price_less_than_intrinsic(
      RealT price
    , RealT fwd
    , RealT strike
    , RealT t
    , e_call_put cp
    , unsigned long its
    , RealT eps
  )
  {
    typedef typename
      result_of::jaeckel_method_err<RealT>::type return_type;

    throw std::runtime_error("price less than intrinsic value");

    return return_type();
  }

};

struct jaeckel_method_default_err_handlers
  : jaeckel_method_default_err_handler_impl
{
  typedef jaeckel_method_default_err_handler_impl base_t;

  typedef base_t err_hnd1_t; //negative fwd.
  typedef base_t err_hnd2_t; //negative strike
  typedef base_t err_hnd3_t; //negative time
    typedef base_t err_hnd4_t; //put price > strike
    typedef base_t err_hnd5_t; //call price > fwd.
    typedef base_t err_hnd6_t; //price < intrinsic
  };

  struct jaeckel_method_err_defaults
    : virtual jaeckel_method_default_err_handlers
  {};

}//namespace detail

template <class P>
struct negative_forward
  : virtual detail::jaeckel_method_default_err_handlers
{
  typedef P err_hnd1_t;
};

template <class P>
struct negative_strike
  : virtual detail::jaeckel_method_default_err_handlers
{
  typedef P err_hnd2_t;
};

template <class P>
struct negative_time_to_maturity
  : virtual detail::jaeckel_method_default_err_handlers
{
  typedef P err_hnd3_t;
};

template <class P>
struct put_price_greater_than_strike
  : virtual detail::jaeckel_method_default_err_handlers
{
  typedef P err_hnd4_t;
};

template <class P>
struct call_price_greater_than_forward
  : virtual detail::jaeckel_method_default_err_handlers
{
  typedef P err_hnd5_t;
};

template <class P>
struct price_less_than_intrinsic
  : virtual detail::jaeckel_method_default_err_handlers
{
  typedef P err_hnd6_t;
};

namespace detail
{
  template <class RealT>
  RealT normalized_black_call(RealT x, RealT sig)
  {
    using std::exp;
    using std::abs;
    using std::pow;

    RealT zero  =0;
    RealT one   =1;
    RealT two   =2;
    RealT three =3;
    RealT four  =4;
    RealT six   =6;
    RealT eight =8;
    RealT eps=boost::math::tools::epsilon<RealT>();
    RealT pi=boost::math::constants::pi<RealT>();
    RealT one_div_sqrt_two_pi=one/sqrt(2*pi);

    RealT x2=x*x;
    RealT s2=sig*sig;
    RealT b_max=exp(0.5*x);
    RealT one_over_b_max=one/b_max;

    if((x2 < eps*s2) || ((x2 + s2) < eps))
    {
      RealT b0 = (s2*s2 > eps)
        ? one - two*norm_cdf(-0.5*sig)
        : one_div_sqrt_two_pi*sig*(
              one - s2*(one/RealT(24)
                          - s2*(one/RealT(640)
                                 - s2*(one/RealT(21504)
                                        - s2/RealT(884736)))));
      return (std::max)(b0 + 0.5*x, zero);
    }

    RealT xi=x/sig;
    if(s2 < eps*x2)
    {
      RealT xi2=xi*xi;
      RealT phi0=exp(-0.5*xi2)*one_div_sqrt_two_pi;

      return (std::max)(
          phi0*exp(-0.125*s2)*four*
              sig/pow(4*xi2 - s2, three)*
```

```
              (eight*xi2*(two*xi2 - s2 - six) + s2*(s2 - four))
          , zero);
      }

      return (std::max)(
           norm_cdf(xi + 0.5*sig)*b_max -
             norm_cdf(xi - 0.5*sig)*one_over_b_max
         , zero);
    }

    template <class RealT>
    inline RealT sig_lo(RealT x, RealT beta, RealT b_c)
    {
      using std::abs;
      using std::log;
      using std::sqrt;

      return sqrt(2.0*x*x/(abs(x) - 4.0*log((beta)/(b_c))));
    }

    template <class RealT>
    inline RealT sig_hi(RealT x, RealT beta, RealT b_c)
    {
      using std::exp;
      using std::sqrt;
      using std::abs;

      RealT e = exp(0.5*x);
      return -2.0*inverse_norm_cdf(((e - beta)/(e - b_c))*
                                      norm_cdf(-sqrt(0.5*abs(x))));
    }

    template <class RealT>
    inline RealT w(RealT xi, RealT gamma)
    {
      using std::pow;

      return (std::min)(pow(xi, gamma), RealT(1.0));
    }

  }//namespace detail

//avoid local code repetition
# define CPPF_JAECKEL_METHOD_ENFORCE(cond, handler, which) \
  if(!(cond))\
    {\
      boost::fusion::tie(status_, vol_, iterations_ ) = \
        handler:: which(\
                    price\
                  , forward\
                  , strike\
                  , time_to_maturity\
                  , call_put_code\
                  , max_its\
                  , eps);\
      return;\
    }\
/**/

  template<
     class RealT
   , class E1
   , class E2
   , class E3
   , class E4
   , class E5
   , class E6
  >
  template <class ItsF, class EpsF>
  jaeckel_method<RealT, E1, E2, E3, E4, E5, E6>::jaeckel_method(
      RealT price
    , RealT forward
    , RealT strike
    , RealT time_to_maturity
    , e_call_put call_put_code
    , ItsF its_f
    , EpsF eps_f)
    : status_(undetermined)
    , vol_(boost::math::tools::max_value<RealT>())
    , iterations_(0ul)
  {
    unsigned long max_its = its_f();
    RealT eps = eps_f();

    //'By Implication', Peter Jaeckel, Oct. 2006

    typedef boost::mpl::vector<E1,E2,E3,E4,E5,E6> policies_t;
    typedef meta::policy_selector<policies_t> err_hnds_t;
    typedef typename err_hnds_t::err_hnd1_t err_hnd1_t; //negative fwd.
    typedef typename err_hnds_t::err_hnd2_t err_hnd2_t; //negative strike
    typedef typename err_hnds_t::err_hnd3_t err_hnd3_t; //negative time
    typedef typename err_hnds_t::err_hnd4_t err_hnd4_t; //put price > strike
    typedef typename err_hnds_t::err_hnd5_t err_hnd5_t; //call price > fwd.
    typedef typename err_hnds_t::err_hnd6_t err_hnd6_t; //price < intrinsic

    using namespace ::cppf::maths::process::lognormal::implied_vol::detail;

    RealT p = price;
    RealT F = forward;
    RealT K = strike;
    RealT T = time_to_maturity;
    RealT theta = call_put_code;
    RealT pi = boost::math::constants::pi<RealT>();
    RealT zero=0;

    CPPF_JAECKEL_METHOD_ENFORCE(
        F > 0
      , err_hnd1_t, on_negative_forward);
    CPPF_JAECKEL_METHOD_ENFORCE(
        K > 0
      , err_hnd2_t, on_negative_strike);
    CPPF_JAECKEL_METHOD_ENFORCE(
        T > 0
      , err_hnd3_t, on_negative_time_to_maturity);
    CPPF_JAECKEL_METHOD_ENFORCE(
        theta ==  1 || price < strike
      , err_hnd4_t, on_put_price_greater_than_strike);
    CPPF_JAECKEL_METHOD_ENFORCE(
        theta ==  -1 || price <= forward
      , err_hnd5_t, on_call_price_greater_than_forward);

    RealT intrinsic=(std::max)(theta*(F - K), zero);
    if(p == intrinsic)
    {
      boost::fusion::vector_tie(status_, vol_) =
        boost::fusion::make_vector(determined, zero);

      return;
    }

    RealT beta=(p - intrinsic)/sqrt(F*K);
    CPPF_JAECKEL_METHOD_ENFORCE(
         beta >= 0
      , err_hnd6_t, on_price_less_than_intrinsic);

    using std::log;
    using std::sqrt;
    using std::exp;
    using std::abs;

    //operate on out-of-the-money calls from here
    RealT x = -abs(theta*log(F/K));
    RealT xdiv2 = 0.5*x;

    //initial guess
    RealT sig_c = sqrt(2*abs(x));
    RealT b_c = normalized_black_call(x, sig_c);
    RealT sig0 = 0;
    if(beta < b_c)
    {
      //get hi and lo and do the interpolation
      RealT siglo = sig_lo(x, beta, b_c);
      RealT sighi = sig_hi(x, beta, b_c);
      RealT sigstar = sig_hi(x, RealT(0), b_c);
      RealT bstar = normalized_black_call(x, sigstar);
      RealT siglostar = sig_lo(x, bstar, b_c);
      RealT sighistar = sig_hi(x, bstar, b_c);

      RealT log_arg1 = (sigstar - siglostar)/(sighistar - siglostar);
      assert(log_arg1 > 0.0);
      RealT log_arg2 = bstar/b_c;
      assert(log_arg2 > 0.0);

      RealT gamma = log(log_arg1)/log(log_arg2);
      RealT t = w(beta/b_c, gamma);
      sig0 = (1.0 - t)*siglo + t*sighi;
      if(normalized_black_call(x, sig0)
           <  boost::math::tools::min_value<RealT>())
      {
        sig0 += sigstar;
        sig0 *= 0.5;
        if(normalized_black_call(x, sig0)
             < boost::math::tools::min_value<RealT>())
        {
          sig0 += sig_c;
          sig0 *= 0.5;
        }
      }
    }
    else
      sig0 = sig_hi(x, beta, b_c);

    RealT sqrt_two_pi = sqrt(2*pi);

    //halley's method
    while(iterations_ < max_its)
    {
      RealT b = normalized_black_call(x, sig0);
      RealT xdivsig2 = (x/sig0)*(x/sig0);
      RealT sigdiv2 = 0.5*sig0;
      RealT sigd2s = (sigdiv2)*(sigdiv2);
      RealT bp = exp(-0.5*xdivsig2 - 0.5*sigd2s)/sqrt_two_pi;

      RealT vn = 0;
      if(beta < b_c)
      {
        vn = log(beta/b)*(log(b)/log(beta))*(b/bp);
      }
```

```
      else
      {
        vn = (beta - b)/bp;
      }
      RealT b2divb1 = x*x/pow(sig0, 3) - 0.25*sig0;
      RealT f2divf1 = b2divb1 -
        (((2 + log(b))/log(b))*bp/b)*((beta < b_c) ? 1 : 0);
      RealT vhatn = (std::max)(vn, -0.5*sig0);
      RealT nhatn = (std::max)(0.5*vhatn*f2divf1, RealT(-0.75));
      RealT sig1 = sig0 + (std::max)(vhatn/(1 + nhatn), -0.5*sig0);

      assert(boost::math::isfinite(sig1));

      if(abs(((sig1/sig0 - 1))) <= eps)
      {
        break;
      }

      sig0 = sig1;
      ++iterations_;
    }

    boost::fusion::vector_tie(status_, vol_) =
      boost::fusion::make_vector(
          iterations_ < max_its ?
          converged : max_iterations_exceeded
        , sig0/sqrt(T)
      );

    return;
  }

#undef CPPF_JAECKEL_METHOD_ENFORCE

}//namespace implied_vol

}}}}//namespace cppf::maths::process::lognormal

#endif//!defined(JAECKEL_METHOD_DDAE8974_C6E8_40B3_AD3A_43417A3B1CAC_INCLUDED)
```

6 Acknowledgements

The author gratefully acknowledges Peter Jaeckel, Peter Bartlett, Christopher Gardner, Lee Wild and Aleksandar Lucic for their thoughtful reviews of this work and their many helpful comments. In particular Peter Jaeckel suggested numerous improvements to, and tests for, the numerical algorithm implementation.

References

Abrahams & Gurtovoy. *C++ Template Metaprogramming*. Addison Wesley, 2005.

Alexandrescu. *Modern C++ Design*. Addison Wesley, 2001.

Black and Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy, 81:637-654*, 1973.

Jaeckel. By implication. *Wilmott Magazine, 60-66*, 2006.

Merton. Theory of rational option pricing. *Bell Journal of Economics and Management Science, 4:141-183*, 1973.

Vandervoorde & Josuttis. *C++ Templates The Complete Guide*. Addison Wesley, 2003.