QuantLib Erlkönige

Peter Caspers

IKB

December 4th 2014

Erlkönig

Table of contents

No Arbitrage SABR - the model

Paul Doust, No-arbitrage SABR, Journal of Computational Finance, Volume 15 / Number 3, Spring 2012. Main Features:

1. approximates the density (with a positive function), thereby producing an arbitrage free smile over strike range $[0, \infty)$
2. assumes arbsorbing barrier at $F = 0$ and reproduces precomputed arbsorption probabilities generated by a MC simulation (published by Paul Doust as well)
3. call prices are computed by numerical integration, implied volatilities are computed by inverting the Black formula

No Arbitrage SABR Example

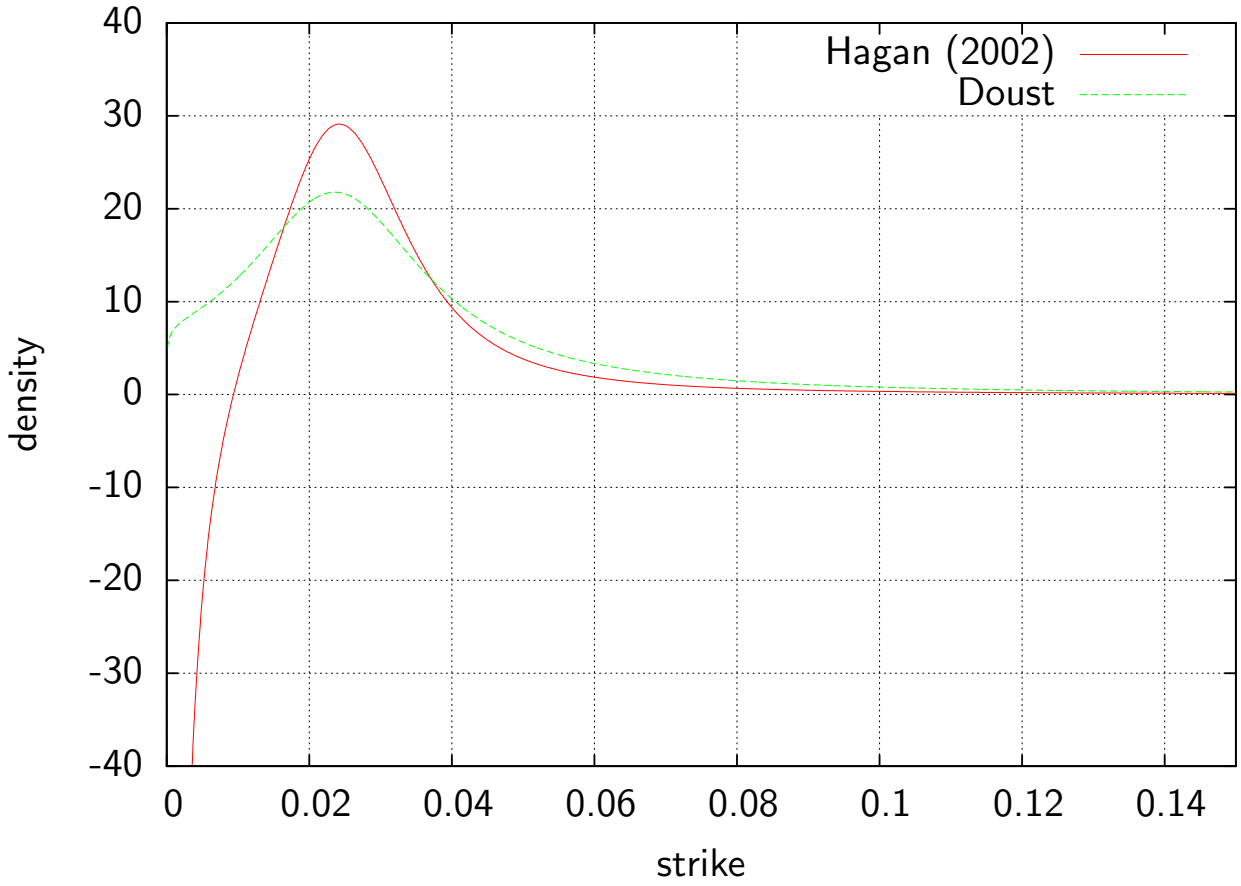


Figure : SABR smile $\alpha = 0.02$, $\beta = 0.40$, $\nu = 0.30$, $\rho = 0.30$, $\tau = 30.0$, $f = 0.03$

No Arbitrage SABR classes

`ql/experimental/volatility/`

`NoArbSabrModel`	core computation formulas
`NoArbSabrInterpolation`	interpolation class
`NoArbSabrSmileSection`	smile section by parameters
`NoArbSabrInterpolatedSmileSection`	interpolating smile section
`SwaptionVolcube1a`	volatility cube

Design changes

Make the SABR interpolation and volatility cube classes generic, so that both models (and possibly more like SVI, ZABR) are accepted. The old `SwaptionVolCube1` class e.g. is now retrieved by

```
struct SwaptionVolCubeSabrModel {
        typedef SABRInterpolation Interpolation;
        typedef SabrSmileSection SmileSection;
};

typedef SwaptionVolCube1x<SwaptionVolCubeSabrModel>
                                        SwaptionVolCube1;
```

and likewise for the new "1a"-variant of the cube using the noarb-SABR formula.

NoArbSABR - limitations

1. there are examples of parameters $(\alpha, \beta, \nu, \rho)$ for which the recalibration of the model implied forward does not work
2. the implied volatility (since inverted from call prices) is not smooth for far otm strikes in some cases (but actually rarely needed because calls, digitals and the density is directly available !)
3. in general, never underestimate the benefit of a pure closed form formula (i.e. Hagan 2002) over a computation involving numerical procedures ...

ZABR, SVI

There are other models fitting in this framework like Andreasen's ZABR model

$$dF = F^{\beta}\sigma dW \tag{1}$$
$$d\sigma = \nu\sigma^{\gamma}dV \tag{2}$$
$$dV\,dW = \rho dt \tag{3}$$

with an additional parameter γ giving more flexibility for wing calibration to e.g. CMS quotes.

ZABR Example

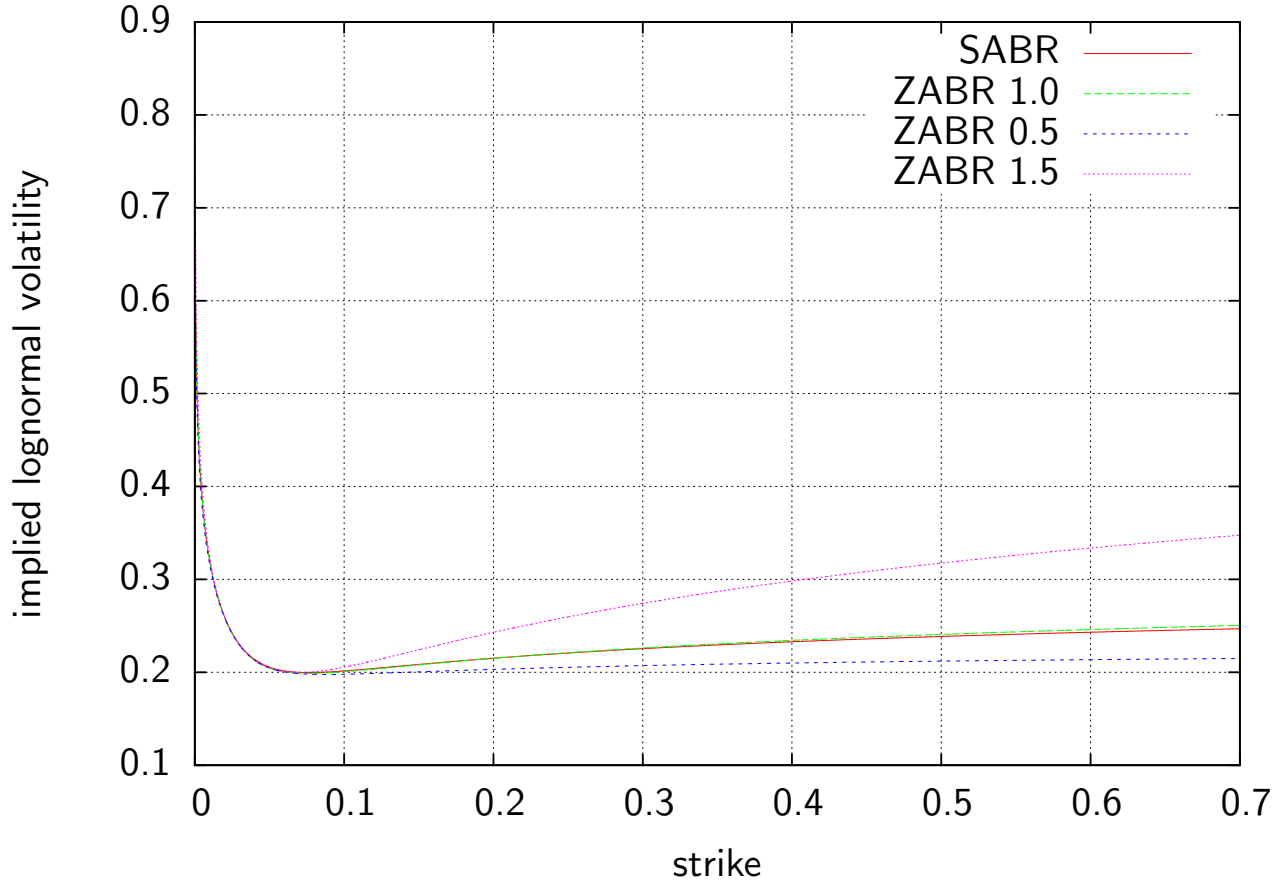


Figure : SABR (Hagan 2002 expansion) $\alpha = 0.03$, $\beta = 0.70$, $\nu = 0.20$, $\rho = -0.30$, $\tau = 5.0$, $f = 0.03$ vs. ZABR (short maturity expansion) for different $\gamma = 0.5, 1.0, 1.5$ controlling the smile wings.

ZABR - more features and limitations

1. two short maturity expansions (normal and lognormal implied volatility)
2. an "equivalent" Dupire - style FD approximation, which is fast and arbitrage free in particular
3. a full finite solution, for benchmarking and testing (slow of course)
4. but ... approximations are not very good for long option expiries
5. advantages for CMS pricing yet to be proved in a productive setting

SVI

SVI is another popular smile model, with the total variance given by

$$v^2 t = a + b\left(\rho(k-m) + \sqrt{(k-m)^2 + \sigma^2}\right) \tag{4}$$

with log moneyness $k = \log K/F$, K = strike, F = forward.

SVI Example

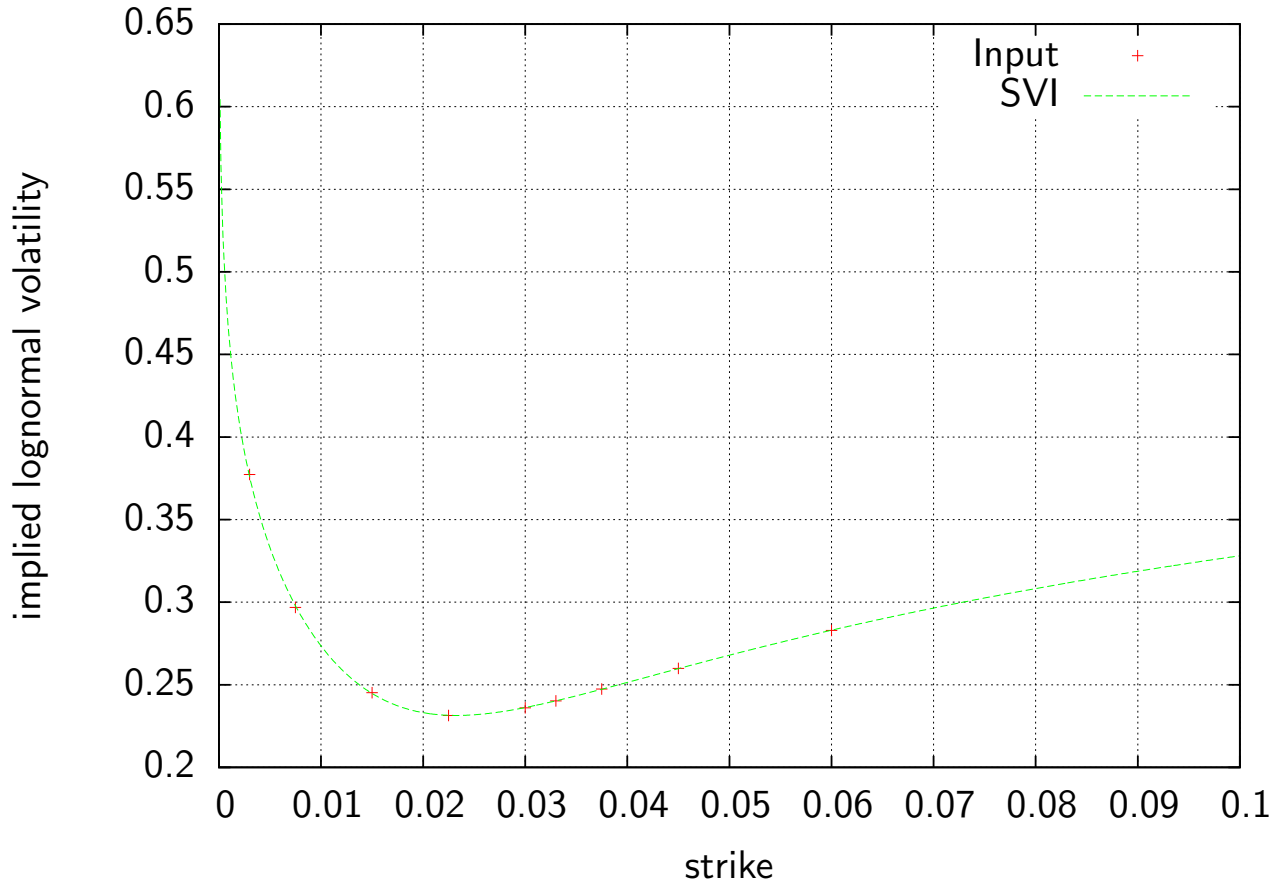


Figure : SVI fit to sample input data generated by SABR ($\alpha = 0.08$, $\beta = 0.90$, $\nu = 0.30$, $\rho = 0.30$, $\tau = 10.0$, $f = 0.03$)

Implementation of SVI 1/2

We can use the generic framework, so e.g. the implementation of the SVI interpolation class is merely specifying the SVI specific things ...

```
typedef SviSmileSection SviWrapper;
struct SviSpecs {
    Size dimension() { return 5; }
    void defaultValues(std::vector<Real> &params, std::vector<bool> &paramIsFixed,
                       const Real &forward, const Real expiryTime) { /* ... */ }
    void guess(Array &values, const std::vector<bool> &paramIsFixed,
               const Real &forward, const Real expiryTime,
               const std::vector<Real> &r) { /* ... */ }
    Array inverse(const Array &y, const std::vector<bool> &,
                  const std::vector<Real> &, const Real) { /* ... */  }
    Array direct(const Array &x, const std::vector<bool> &paramIsFixed,
                 const std::vector<Real> &params, const Real forward) { /* ... */ }
    typedef SviWrapper type;
    boost::shared_ptr<type> instance(const Time t, const Real &forward,
                                     const std::vector<Real> &params) { /* ... */ }
};
```

Implementation of SVI (2/2)

... and use this in the generic implemenation:

```
class SviInterpolation : public Interpolation {
  public:
    template <class I1, class I2>
    SviInterpolation(const I1 &xBegin, ... ) {
        impl_ = boost::shared_ptr<Interpolation::Impl>(
            new detail::XABRInterpolationImpl<I1, I2, detail::SviSpecs>(
                xBegin, xEnd, yBegin, t, forward,
                boost::assign::list_of(a)(b)(sigma)(rho)(m),
                boost::assign::list_of(aIsFixed)(bIsFixed)(sigmaIsFixed)(
                    rhoIsFixed)(mIsFixed),
                vegaWeighted, endCriteria, optMethod, errorAccept, useMaxError,
                maxGuesses));
        coeffs_ = boost::dynamic_pointer_cast<
            detail::XABRCoeffHolder<detail::SviSpecs> >(impl_);
    }
```

Note that `XABR...` is already to narrow as the label for the generic class. Better than the other way round ...

SVI - limitations

1. uses the “raw” parametrization, which does not allow for easy parameter interpretation
2. calibration is naive, i.e. does not avoid local minima / parameter identifcation problem cases

Other smile models, general thoughts

There are other models, that are not fitted, but interpolate given points by construction such that the resulting smile is arbitrage free, e.g.

1. KahaleSmileSection which is already used implicity by the Markov functional model
2. BDK which fixes arbitrageable wings and introduce new parameters for wing calibration

Goal: Integrate them as well in a uniform infrastructure providing

1. an interpolation class
2. a smile section which takes either parameters, market data or a source smile section to be smoothed / made arb-free
3. a swaption volatility cube with the possibility to calibrate to the cms market and a caplet volatility surface

TSR CMS Coupon Pricers

A terminal swap rate (TSR) model is given by a mapping α

$$\alpha(S(t)) = \frac{P(t, t_P)}{A(t)} \tag{5}$$

where t_p is the coupon payment date and $A(t)$ the annuity of the underlying swap rate S. Then (integration by parts) the npv of a general CMS coupon $A(0)E^A(P(t,t_p)A(t)^{-1}g(S(t)))$ is given by

$$A(0)S(0)\alpha(S(0)) + \int_{-\infty}^{S(0)} w(k)R(k)dk + \int_{S(0)}^{\infty} w(k)P(k)dk \tag{6}$$

with t begin the fixing date of the coupon, R and P prices of market receiver and payer swaptions and weights $w(s) = \{\alpha(s)g(s)\}''$.

Hagan non parallel shifts model

In Hagan's classic paper, the model A.4 "non parallel shifts" corresponds to the following choice of α

$$\alpha(S) = \frac{S e^{-|h(t_p)-h(t)|x}}{1 - \frac{P(0,t_n)}{P(0,t)} e^{-|h(t_n)-h(t)|x}} \tag{7}$$

with t_n being the last payment date of the underyling swap and $h(s) - h(t) = \frac{1-e^{-\kappa(s-t)}}{\kappa}$ with a mean reversion parameter κ and x implicitly given by

$$S(t) \sum \tau_j P(0,t_j) e^{-|h(t_j)-h(t)|x} + P(0,t_n) e^{-|h(t_n)-h(t)|x} = P(0,t) \tag{8}$$

with τ_j, t_j being the yearfractions and payment dates of the fixed leg of the underlying swap.

Linear TSR model

The linear terminal swap rate model is defined by

$$\alpha(S) = as + b \tag{9}$$

b is determined by the no arbitrage condition

$$P(0, t_p)/A(0) = E^A(P(t, t_p)/A(t)) = aS(0) + b \tag{10}$$

a can be specified indirectly via a reversion κ by setting

$$a = \frac{\partial}{\partial S(t)} \frac{P(t, t_p)}{A(t)} \tag{11}$$

and evaluating the r.h.s. within a one factor gaussian model.

Put Call Parity Example

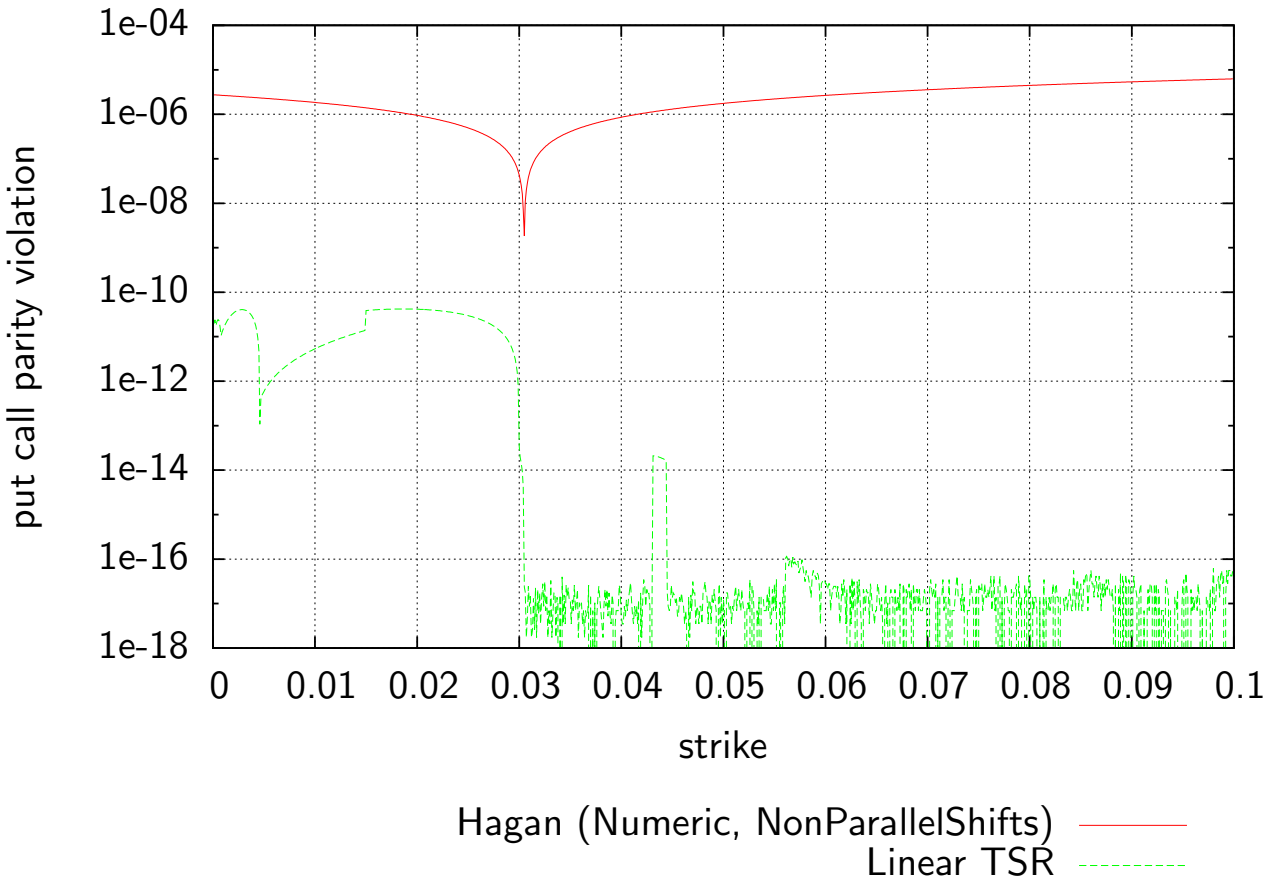


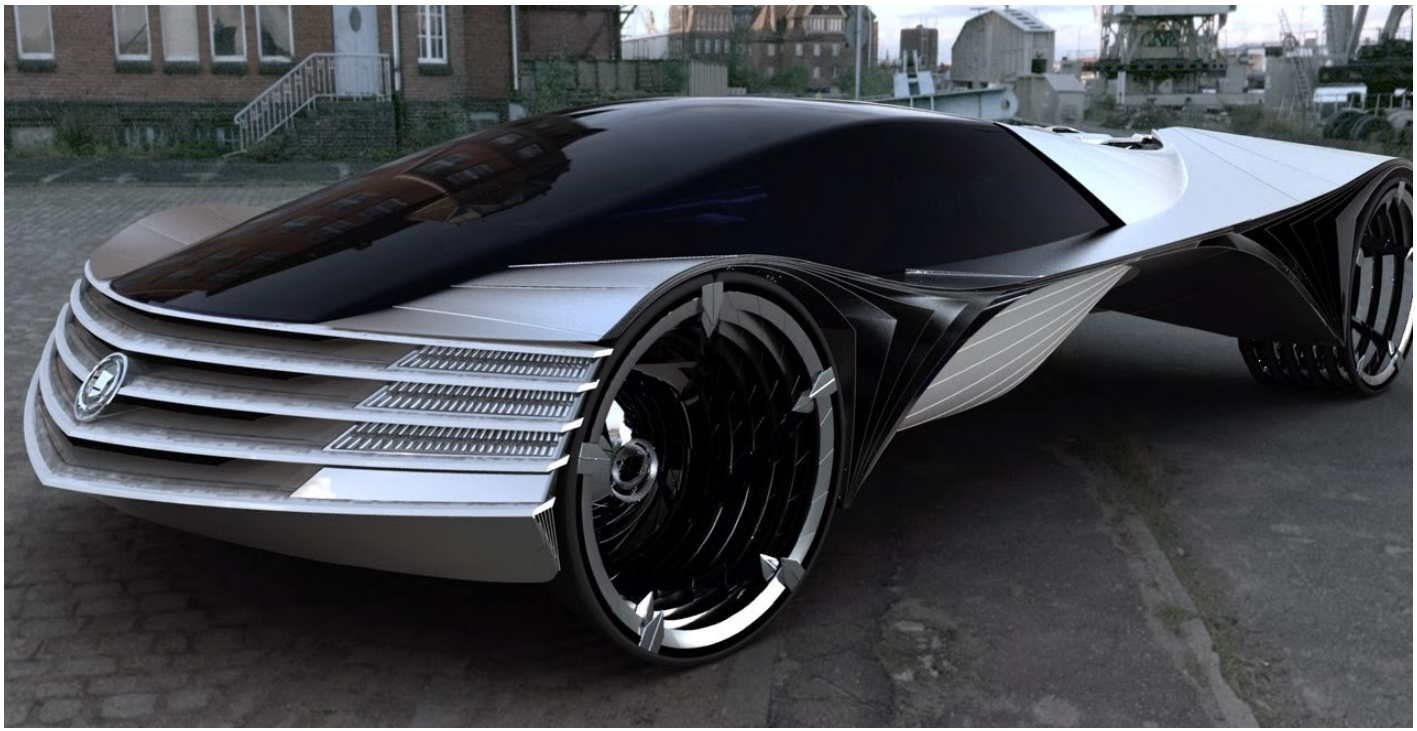
Figure : Parity error for a CMS10y coupon with in arrears fixing in 10y from today, Forward is 0.03, Volatility is given by a SABR surface with $\alpha = 0.10$, $\beta = 0.80$, $\nu = 0.40$, $\rho = -0.30$, reversion is zero, Integration Accuracy for the Linear TSR pricer is 10^{-10}

Performance

Computation time for pricing of 4000 optionlets (parameters as before) on Intel(R) Core(TM) i7-2760QM CPU @ 2.40GHz, single threaded.

`NumericHaganPricer`(Standard)	1840ms
`NumericHaganPricer`(NonParallelShifts)	5882ms
`AnalyticHaganPricer`(Standard)	770ms
`AnalyticHaganPricer`(NonParallelShifts)	741ms
`LinearTsrPricer`	505ms

Fast Erlkönig



CMS Spread Coupons

Still missing: a coupon class which models cms spread coupons

$$\tau(\text{CMS10y} - \text{CMS2y}) \tag{12}$$

possibly capped and / or floored.

Approach 1: Formula index

Introduce an artificial index derived from `InterestRateIndex`

```
SwapSpreadIndex(const std::string& familyName,
                const boost::shared_ptr<SwapIndex>& swapIndex1,
                const boost::shared_ptr<SwapIndex>& swapIndex2,
                const Real gearing1 = 1.0,
                const Real gearing2 = -1.0);
```

and build everything else on top of it as with the other coupons based on ibor or cms indexes.

Approach 1: Repairing the class hiearchy

Since the formula index does not have own fixings, we would have to adjust the index base class by adding

```
//! check if index allows for native fixings
virtual void checkNativeFixingsAllowed() {}
```

and forbid native fixings in formula based indices

```
//! check if index allows for native fixings
virtual void checkNativeFixingsAllowed() {}
void checkNativeFixingsAllowed() {
    QL_FAIL("native fixings not allowed in swap spread index, refer to "
            "underlying indices instead");
}
```

A nicer solution would be to make the addFixing methods virtual and throw an exception in CMS spread index, but they are template methods.

Approach 2: Construct coupons with two swap indexes

If two swap indexes are used to construct a cms spread coupon we would need a more flexible way to construct floating legs, since

```
template <typename InterestRateIndexType,
          typename FloatingCouponType,
          typename CappedFlooredCouponType>
Leg FloatingLeg(const Schedule& schedule,
                const std::vector<Real>& nominals,
                const boost::shared_ptr<InterestRateIndexType>& index,
                const DayCounter& paymentDayCounter,
                BusinessDayConvention paymentAdj,
                const std::vector<Natural>& fixingDays,
                const std::vector<Real>& gearings,
                const std::vector<Spread>& spreads,
                const std::vector<Rate>& caps,
                const std::vector<Rate>& floors,
                bool isInArrears, bool isZero) {
```

only allows for one index.

Approach 2: Coupon Factories

We could introduce a factory instead of the template parameters

```
Leg FloatingLeg(const FloatingCouponFactory& factory,
                const Schedule& schedule,
                ...
```

which can generate plain, capped / floored and digital couons for the ibor, cms, cms spread flavours.

```
  class FloatingCouponFactory {
    virtual boost::shared_ptr<FloatingRateCoupon>
      plainCoupon(const Date &paymentDate, Real nominal,...)
    virtual boost::shared_ptr<CappedFlooredCoupon>
      cappedFlooredCoupon(const Date &paymentDate, Real nominal,...)
    virtual boost::shared_ptr<DigitalCoupon> digitalCoupon(
      const Date &paymentDate, Real nominal, const Date &startDate,...)
    virtual Natural defaultFixingDays() const = 0;
};
```

CMS Spread Coupons - Summary

- Introducing a formula based index would not exactly fit the semantics of the Index class. We would have to distinguish between native indexes (with own fixings) and derived ones. On the other hand this seems to be a quite generic approach, since formula based indexes could be used whereever an InterestRateIndex is allowed
- Using two indexes in the spread coupon class forces us to introduce a more flexible way to construct floating legs, e.g. via factories. This keeps the design clean and the semantics of index sharp. However this is not 100% backward compatible since FloatingLeg is in the main QuantLib namespace.

Credit Risk Plus

A single period, nominal based credit portfolio model, based on Credit Risk Plus, with some extensions allowing for correlated sectors (Integrating Correlations, Risk, July 1999).

```
CreditRiskPlus(const std::vector<Real> &exposure,
    const std::vector<Real> &defaultProbability,
    const std::vector<Size> &sector,
    const std::vector<Real> &relativeDefaultVariance,
    const Matrix &correlation, const Real unit);
```

The loss distribution is computed analytically, so very fast. The model comes with a decomposition of the unexpected loss into single obligors' marginal losses.

Gaussian1d Models

Framework for one factor models with the following interface

```
virtual const Real numeraireImpl(const Time t, const Real y,
                                 const Handle<YieldTermStructure> &yts) const = 0;
virtual const Real zerobondImpl(const Time T, const Time t,
                                const Real y,
                                const Handle<YieldTermStructure> &yts) const = 0;
```

Currently two instances exist

1. `MarkovFunctional`, a non parametric Markov functional model with piecewise volatility and constant reversion
2. `Gsr`, a Hull White model with piecewise volatility and piecewise reversion

Gaussian1d Models - Features

1. multi-curve enabled
2. engines for standard swaptions, swaptions with non-constant nominal, rates, float-float swaptions
3. engines inherit from `BasketGeneratingEngine` that can generate calibration baskets by npv-delta-gamma matching
4. engines take an OAS allowing for exotic bond valuation

Simulated Annealing

A global optimizer based on Nelder-Mead and additional noise in the target function.

1. the noise is exponentially distributed with parameter $1/T$ ("Temperature"), i.e. the expectation and the standard deviation of the noise is both T
2. the optimization starts with a temperature $T > 0$ which decreases to zero during the optimization
3. if the start temperature is high enough and the decrease is slow enough, a global minimum is found with probability one

Global optimization test function

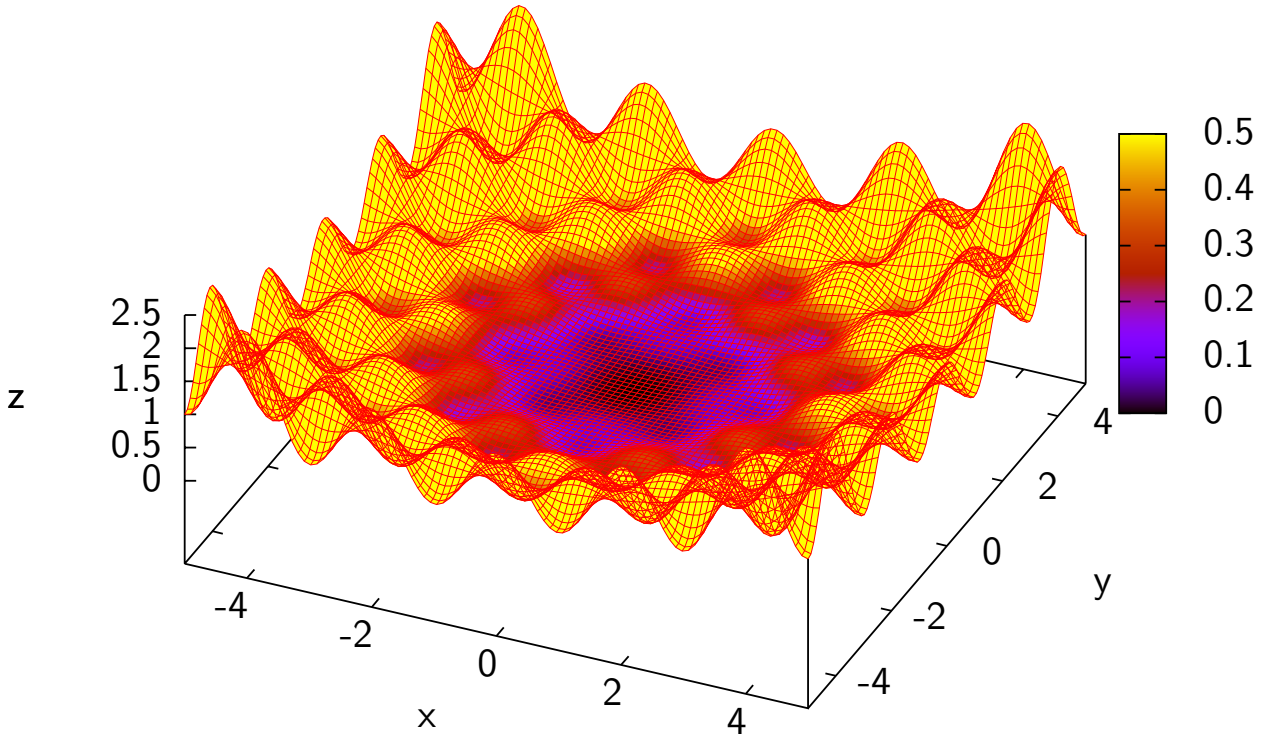


Figure : test function for global optimization $\frac{\{\sin(\pi(x+\frac{1}{2}))\cos(\pi(y+1))+2\}(x^2+y^2)}{50}$

Comparison of optimizers

Optimizer	found minimum	target fct	#evaluations
`Simplex`	(2.965, 2.965)	0.356	128
`SimulatedAnnealing`	(0.0, 0.0)	0.0	10962
`DifferentialEvolution`	(0.0, 0.0)	0.0	500700

1. Nelder-Mead lambda $= 0.2$
2. Start Temperature for simulated annealing is $T = 1.0$ and decreased by a factor of 0.9 each 5 optimization steps
3. Configuration for DE is the default one

Runge Kutta ODE Solver

An ODE solver using a forth order Runge Kutta scheme with adaptive step size control (as described in Numerical Recipes in C, Chapter 17.2). It integrates

$$\frac{d}{dx} f = F(x, f(x)) \tag{13}$$

over an interval $[a, b]$ for $f : [a, b] \to \mathbb{K}^n$ with $\mathbb{K}$ denoting the real or complex numbers with initial condition $f(a) = f_a$.

Solving the modified Bessel equation

As an example we solve the modified Bessel equation

$$x^2y'' + xy' - (x^2 + \alpha)y = 0 \tag{14}$$

for $\alpha = 1$ with $y(0) = 0$ and $y'(0) = 0.5$ over $[0, 10]$ and compare it to the expected result $I_\alpha(10)$ (`modifiedBesselFunction_i`).

Reduction to first derivatives

Equation 14 is equivalent to the system

$$y' = z \tag{15}$$

$$x^2 z' + xz - (x^2 + \alpha)y = 0 \tag{16}$$

with $y(0) = 0$ and $z(0) = y'(0) = 0.5$.

Code example for ODE solving

The ODE F can be defined as follows:

```
Disposable<std::vector<Real>> rhs(const double x,
                                  const std::vector<Real> &f) {
    std::vector<Real> result(2);
    result[0] = f[1];
    if (close(x, 0.0))
        result[1] = (2.0 + alpha * alpha) * f[0] / 2.0;
    else
        result[1] = ((x * x + alpha * alpha) *
                     f[0] - x * f[1]) / (x * x);
    return result;
}
```

Code example for ODE solving

To compute $I_1(10)$ we can then write

```
AdaptiveRungeKutta<Real> rk( 1e-16 );
std::vector<Real> y;
y += 0.0, 0.5;
Real i1 = rk( rhs, y , 0.0, 10.0 )[0];
```

with an ultra-tight tolerance here, just to see what is possible.

ODE solution vs. semi-analytical solution

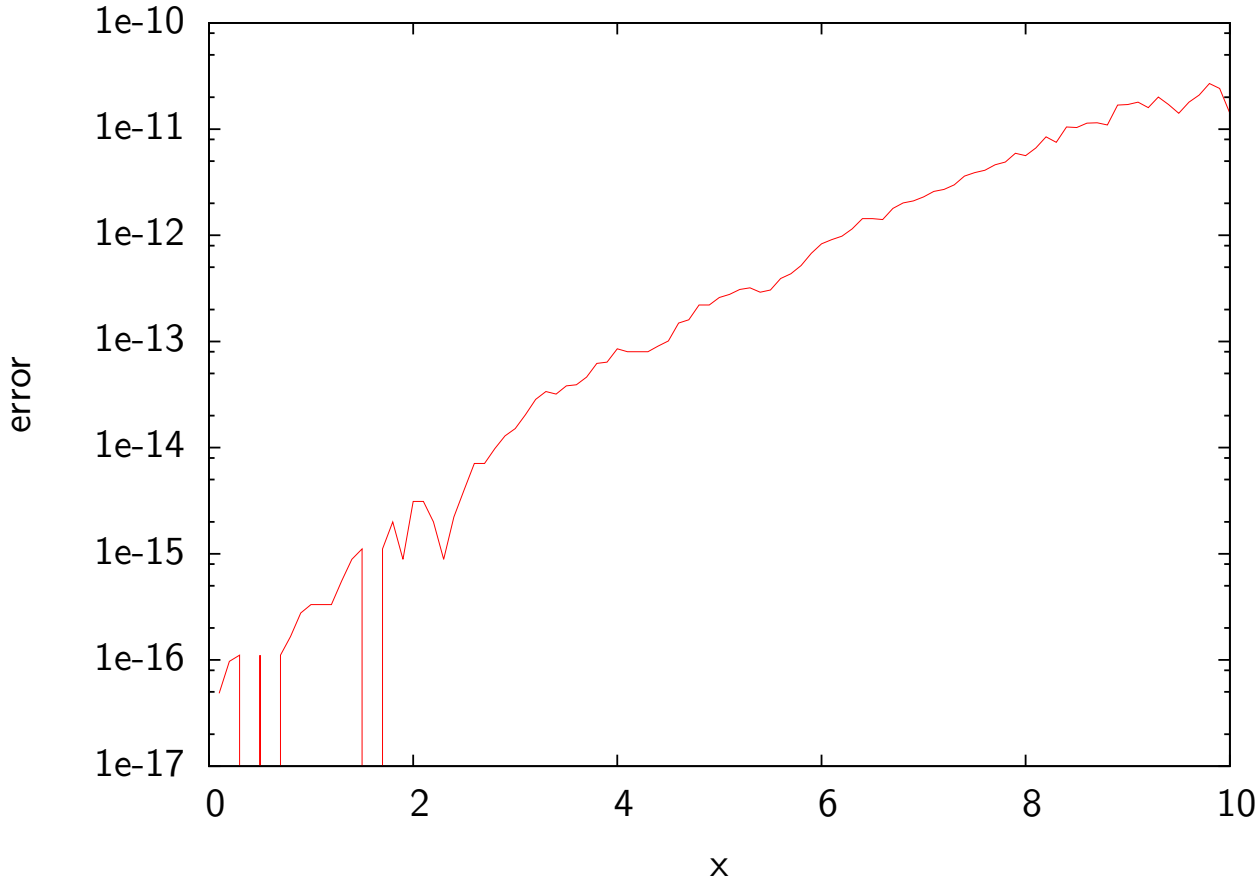


Figure : Runge Kutta adaptive step size solution with $\epsilon = 10^{-16}$ vs. modifiedBessel_i

Dynamic Creator of Mersenne Twister

Makoto Matsumoto and Takuji Nishimura, "Dynamic Creation of Pseudorandom Number Generators" and their implementation as a C library (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html) addresses two needs

1. create Mersenne Twister instances with smaller state space than the classic instance (624 words of 32 bit) and smaller period than $2^{19937} - 1$, or bigger ones if you really want ...
2. create "independent" Mersenne Twister intances for different id's for use in parralel monte carlo

The QuantLib wrapper support both dynamic creation of instances (`MersenneTwisterDynamicRng`) as well as the usage of precomputed instances (`MersenneTwisterCustomRng<Description>`).

Instantiation of dynamic MT's

Dynamically create an instance with 32 bit word size, $p = 521$, creator seed 123, id 0 and seed 42:

```
MersenneTwisterDynamicRng mt(32,521, 123, 0, 42);
```

Use a precomputed instance with seed 42 (p is 19937 here, id is 0)

```
MersenneTwisterCustomRng<Mtdesc19937_0> mt(42);
```

The second alternative is much faster in random number generation. Also it takes a long time to dynamically create MT instances for bigger p.

Example: Parallel RNG streams

This code computes π using parallel mc with 8 threads

```
#define BOOST_PP_LOCAL_LIMITS (0, 7)
#define BOOST_PP_LOCAL_MACRO(n) MersenneTwisterCustomRng<Mtdesc19937_##n> mt##n(42);
#include BOOST_PP_LOCAL_ITERATE()
    omp_set_num_threads(std::min(8, omp_get_max_threads()));
    Real sum = 0.0;
    Size N = 1E8;
#pragma omp parallel for reduction(+ : sum) schedule(static)
    for (Size i = 0; i < N; ++i) {
        Size thread = omp_get_thread_num();
        Real u=0.0,v=0.0;
#define BOOST_PP_LOCAL_LIMITS (0, 7)
#define BOOST_PP_LOCAL_MACRO(n) if(thread==n) { u=mt##n.nextReal(); v=mt##n.nextReal(); }
#include BOOST_PP_LOCAL_ITERATE()
        if(u*u+v*v <= 1.0) sum+=1;
    }
    std::cout << std::setprecision(8) << 4.0 * sum / N << std::endl;
```

Actually this does not run faster multithreaded due to compiler optimizations (vectorization) of the loop, nevertheless illustrates how to use it.

What does "independent" mean ? (1/2)

The MT sequence can be seen as a recurrence

$$s_n = As_{n-1} \tag{17}$$
$$x_n = Bs_n \tag{18}$$

with a state transition matrix A and an output transformation matrix B. s_n satisfies the following equation in $\mathbb{F}_2^k$, k being the bit size of the state space

$$\chi(A)s_{n-k} = 0 \tag{19}$$

with the characteristic polynomial χ of A (Cayley Hamilton Theorem).

What does "independent" mean ? (2/2)

Two independent MT instances have by definition coprime characteristic polynomials f, g, thus there is an isomorphism of the residual polynomial rings (thanks to the chinese remainder theorem)

$$\mathbb{F}_2[T]/(fg) \cong \mathbb{F}_2[T]/(f) \times \mathbb{F}_2[T]/(g) \tag{20}$$

which is formalizing what independdence of the recurrences of the two MT instances mean.

Questions / Discussion

French Erlkönig