

## [Help](#)

```
/*-----*/
/*  Monte Carlo algorithm for caplet prices in one-factor LMM with jumps  */
/*  Algorithm of Glasserman/Merener                                         */
/*                                                                           */
/*-----*/
/*  Sonke Blunck, Premia 2005                                             */
/*-----*/

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2007+2) //The "#els
#else

extern "C" {
#include "
href../../mod/lmm_jump1d/lmm_jump1d_stdh/lmm_jump1d_stdh_src.pdfhmm_jump1d_
}
#include <iostream>
#include <cmath>
#include "
href../../common/math/lmm/math_andersen_h_src.pdfmath/lmm/math_andersen.h"
#include "
href../../mod/lmm_jump1d/lmm_jump1d_stdh/glassermanmerener_h_src.pdfglasserma

using namespace std;

double LN_density(double x, double m = 0., double v = 1.)
// up to the multiplicative factor M_1_SQRT2PI/sqrt(v),
// this is the standard LogNormal density (corresp. to mean m
// and var. v)
{
    return exp(-SQR(m - log(x)) / (2 * v)) / x;
}

GlassMer::GlassMer(double delta, double L0, double gamma, double h, int M):
    _delta(delta), _gamma(gamma), _h(h), _M(M)
{
    int i, k;
```

```

_sqrt_h = sqrt(_h);
_psi_factor = SQR(_gamma) * _delta;

//_x0 = 0.00208; _x1 = 65.1186;
// outside [x0,x1] the lognormal density is < 1e-06 !!
// _x0 = 0.00135; _x1 = 99.8251;
// outside [x0,x1] the lognormal density is < 1e-07 !!

_x0 = 0.0001;
_x1 = 100.;
_DeltaX = 0.05; // should be: 0.05

_a_ctr_max = 20;

_T.resize(_M + 1);
_L0.resize(_M + 1);
_Lt.resize(_M + 1);
_lambda.resize(_M);
_sigma.resize((_M + 1)*M);
_DeltaJ.resize(_M + 1);
_a.resize(_M + 1);
_H.resize(_M + 1);

// init. of _T
for (i = 0; i <= _M; i++)
{
    _T[i] = i * _delta;
}

// init. of _L0
for (i = 0; i <= _M; i++)
{
    _L0[i] = L0;
}

// init. of _lambda
_lambda[0] = 5.0;
for (k = 1; k < _M; k++)

```

```

    {
        _lambda[k] = _lambda[k - 1] * 0.99;
    }

    // init. of _sigma
    for (i = 0; i <= _M; i++)
    {
        _sigma[i * _M] = 0.1;
        for (k = 1; k < _M; k++)
        {
            _sigma[i * _M + k] = _sigma[i * _M + k - 1] * 1.01; // = sigma_{i,k}
        }
    }

} // end of the constructor


void GlassMer::InitialCond(int generator)
{
    _Lt = _L0;
    _t = 0.;
    _eta = 1;
    _eta_old = 1;
    _a_ctr = _a_ctr_max;
    _Xi = -log(pnl_rand_uni(generator));
}


int GlassMer::eta(double t)
// returns the index k such that t is in (_T[k-1],_T[k]]
{
    if (t > 0) return (int)ceil(t / _delta);
    else return 1;
}


void GlassMer::Set_t(double t)
{
    _t = t;
    _eta_old = _eta;
}

```

```

_eta = eta(t);

_a_ctr += 1;
if (_eta == _eta_old + 1) _a_ctr = _a_ctr_max;

if (fabs(_h * pnl_iround(_t / _h) - _t) > 0.01)
{
    cout << "_t-alert !" << endl;
    exit(1);
}
}

double GlassMer::H(int i, double x, double t)
{
    return pow(x, _sigma[i * _M + eta(t) - 1]) - 1.;
} // = sigma_{i,eta(t)-1}

void GlassMer::Set_H(double x)
{
    for (int i = _eta; i <= _M; i++) _H[i] = H(i, x, _t);
}

double GlassMer::phi(int i)
// returns phi_i(_t,_H,_Lt) as in the documentation
{
    double res = 1.;

    for (int j = _eta; j <= i; j++)
        res *= (1 + _delta * _Lt[j]) / (1 + _delta * _Lt[j] * (1 + _H[j]));

    return res;
}

double GlassMer::psi(int i)
// returns psi_i(_t,_Lt) as in the documentation
{
    double sum = 0.;

```

```

    for (int j = _eta; j <= i; j++)
        sum += _Lt[j] / (1 + _delta * _Lt[j]);

    return _psi_factor * sum;
}

double GlassMer::a(int i)
// returns the forward measure drift  $a^i_{\{t\}}$  as in the docum.
{
    double integr = 0.;
    double x = _x0;

    while (x < _x1)
    {
        Set_H(x); // _H[i]=H(i,x,t) for i=_eta,...,_M
        integr -= _H[i] * phi(i) * LN_density(x);
        x += _DeltaX;
    }

    return _lambda[_eta - 1] * M_1_SQRT2PI * _DeltaX * integr;
}

void GlassMer::Set_a(int i0)
{
    if (_a_ctr >= _a_ctr_max)
    {
        for (int i = i0; i <= _M; i++) _a[i] = a(i);
        _a_ctr = 0;
    }
}

double GlassMer::Lambda(double t)
// the function (capital) Lambda in the doc.
{
    double sum = 0.;

    for (int k = 0; k <= eta(t) - 2; k++) sum += _lambda[k];
}

```

```

    return _delta * sum + (t - _T[eta(t) - 1]) * _lambda[eta(t) - 1];
}

```

```

void GlassMer::Scheme(int generator)
// one simulation step (from _t to _t+_h) under the spot measure
{
    int i, jump_flag = 0;
    double DeltaW, X, b;

    DeltaW = _sqrt_h * pnl_rand_normal(generator);
    _DeltaJ = 0.;
    // computation of the jumps _DeltaJ[i]
    while (_Xi <= Lambda(_t + _h))
    {
        jump_flag = 1;
        X = exp(pnl_rand_normal(generator));
        for (i = eta(_t + _h); i <= _M; i++)
            _DeltaJ[i] += H(i, X, _t);
        _Xi -= log(pnl_rand_uni(generator));
    }
    // compute the new (i.e. at time _t+_h) values of _Lt
    // (Observe that _Lt[i] is simulated from 0 to _T[i])
    Set_a(eta(_t + _h));
    for (i = _M; i >= eta(_t + _h); i--)
    {
        b = psi(i) + _a[i]; // the spot measure drift
        _Lt[i] += _Lt[i] * (_gamma * DeltaW + _DeltaJ[i] + b * _h);
    }

    if (jump_flag) _a_ctr = _a_ctr_max;

    Set_t(_t + _h);
}

```

```

double GlassMer::CapletMC(double K, int M, int generator)
// MC simulation of the spot measure dynamics
{
    int l, m, NT;

```

```

double caplet, spot_numeraire, sum = 0., sqr_sum = 0.;

NT = pnl_iround(_T[_M] / _h);

for (m = 0; m < M; m++)
{
    InitialCond(generator);
    for (l = 0; l < NT; l++) Scheme(generator); // now time = _T[_M]

    spot_numeraire = 1. + _delta * _Lt[0];
    for (l = 1; l <= _M; l++) spot_numeraire *= 1. + _delta * _Lt[l];

    caplet = MAX(_Lt[_M] - K , 0.) / spot_numeraire;
    sum += caplet;
    sqr_sum += SQR(caplet);
}

sum *= _delta;
sqr_sum *= SQR(_delta);
//var_estim = (sqr_sum - SQR(sum)/M) / (double)(M-1);
//      cout << "95% conf. interval = " << 10000*1.96*sqrt(var_estim/M)
//              << endl;

return sum / (double)M;
}

```

```

double GlassMer::CapletCF(double K)
// the Glassermann/Merener CF approximation of current caplet price
{
    int i, k;
    double x, u, I, J, Pi1, Pi2, B1, B2, B3, B4, discount, logterm;
    double expterm1, expterm2;

    double u0 = 0.001;          // integration parameters for the
    double u1 = 20.;            // computation of Pi1 and Pi2
    double DeltaU = 0.001;

    double x0 = 0.0001;         // integration parameters

```

```

double x1 = 100.;
double DeltaX = 0.05;

valarray<double> hat_lambda(0., _M); // jump intensity
valarray<double> hat_a(0., _M);      // drift
valarray<double> hat_sigma(_M);      // lognormal parameter
valarray<double> hat_mu(_M);         // lognormal parameter
valarray<double> hat_m(_M);
valarray<double> hat_var(_M);        // for the SQR(hat_sigma[k])
valarray<double> alpha(_M);          // as in the pricing thm
valarray<double> omega(_M);          // as in the pricing thm

_Lt = _L0;

// computation of lambda hat, sigma hat, mu hat, var hat
for (k = 0; k < _M; k++)
{
    _t = _T[k + 1];
    _eta = k + 1;
    I = 0.;
    J = 0.;
    x = x0;

    while (x < x1)
    {
        Set_H(x); // _H[i]=H(i,x,_t) for i=_eta,...,_M
        hat_lambda[k] += phi(_M) * LN_density(x);
        I += _H[_M] * phi(_M) * LN_density(x);
        J += SQR(_H[_M]) * phi(_M) * LN_density(x);
        x += DeltaX;
    }

    hat_lambda[k] *= _lambda[k] * M_1_SQRT2PI * DeltaX;
    I *= _lambda[k] / hat_lambda[k] * M_1_SQRT2PI * DeltaX;
    J *= _lambda[k] / hat_lambda[k] * M_1_SQRT2PI * DeltaX;

    hat_var[k] = log((J + 1 + 2 * I) / SQR(1 + I));
    hat_mu[k] = log(1 + I) - hat_var[k] / 2;
}

```



```

    hat_sigma[k] = sqrt(hat_var[k]);
}

// computation of the drift hat a
for (k = 0; k < _M; k++)
{
    x = x0;

    while (x < x1)
    {
        hat_a[k] -= (x - 1) * LN_density(x, hat_mu[k], hat_var[k]);
        x += DeltaX;
    }

    hat_a[k] *= M_1_SQRT2PI / hat_sigma[k] * DeltaX;
    hat_a[k] *= hat_lambda[k];
}

// computation of m hat, omega, alpha
for (k = 0; k < _M; k++)
{
    hat_m[k]    = exp(hat_mu[k] + hat_var[k] / 2.) - 1.;
    omega[k]    = hat_mu[k] + hat_var[k];
    alpha[k]    = hat_a[k] - SQR(_gamma) / 2.;
}

// computation of Pi1, Pi2
logterm = log(K / _L0[_M]);
Pi1 = 0.;
Pi2 = 0.;
u = u0;

while (u < u1)
{
    B1 = 0.;
    B2 = 0.;
    B3 = 0.;
    B4 = 0.;

```

```

for (k = 0; k < _M; k++)
{
    expterm1 = exp(hat_mu[k] + hat_var[k] * (1 - SQR(u)) / 2.);
    expterm2 = exp(-SQR(hat_sigma[k] * u) / 2.);

    B1 += hat_lambda[k] * (expterm1 * cos(omega[k] * u) - 1.);
    B1 -= hat_lambda[k] * hat_m[k] + SQR(_gamma * u) / 2.;

    B2 += hat_lambda[k] * expterm1 * sin(omega[k] * u) + alpha[k] * u;
    B2 += SQR(_gamma) * u;

    B3 += hat_lambda[k] * (expterm2 * cos(hat_mu[k] * u) - 1.);
    B3 -= SQR(_gamma * u) / 2.;

    B4 += hat_lambda[k] * expterm2 * sin(hat_mu[k] * u);
    B4 += alpha[k] * u;
}

B1 *= _delta;
B2 *= _delta;
B3 *= _delta;
B4 *= _delta;

Pi1 += exp(B1) * sin(B2 - u * logterm) / u;
Pi2 += exp(B3) * sin(B4 - u * logterm) / u;

u += DeltaU;
}

Pi1 = 0.5 + Pi1 * DeltaU / M_PI;
Pi2 = 0.5 + Pi2 * DeltaU / M_PI;

// computation of the result
discount = 1.;
for (i = 0; i <= _M; i++) discount /= 1 + _delta * _L0[i];

// cout << "estimate for K=0 : " << _delta*discount*_L0[_M] << endl;

return _delta * discount * (_L0[_M] * Pi1 - K * Pi2);
}

```

```

int lmm_jump_caplet_GlassMer_pricer(double tenor, double capletMat, double K, double f)
// caplet pricing via the CF approx. method of Glassermann/Merener
{
    int M = pnl_iround(capletMat / tenor);

    GlassMer GM(tenor, flatInitialValue, vol, 0.01, M);

    *price = GM.CapletCF(K);
    return OK;
}

int lmm_jump_caplet_MC_pricer(double tenor, double capletMat, double K, double f)
// caplet pricing via Monte Carlo
{
    int M = pnl_iround(capletMat / tenor);

    GlassMer GM(tenor, flatInitialValue, vol, 0.05, M);

    *price = GM.CapletMC(K, numberMCPaths, generator);
    return OK;
}
#endif //PremiaCurrentVersion

```