

[Help](#)

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

/// \ file cirpp.h
/// \ brief numerical constant
/// \ author M. Ciuca (MathFi, ENPC)
/// \ note (C) Copyright Premia 8 - 2006, under Premia 8 Software license
//
// Use, modification and distribution are subject to the
// Premia 8 Software license

#ifndef _CIRPP_H
#define _CIRPP_H

// The couple of files (cirpp.h, cirpp.cpp) implements the numerical methods
// presented in the paper:
// Brigo D., Alfonsi A. (2004), "Credit Default Swaps calibration and option
// pricing with the SSRD stochastic intensity and interest-rate model"

#include <stdexcept>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <
href../../../../common/math/highdim_solver/highdim_vector_h_src.pdfvector>
#include <
href../../../../common/math/cdo/cdo_math_h_src.pdfmath.h>

//#define NDEBUG
#include <cassert>

#include "
href../../../../common/math/credit_cds/base_h_src.pdfbase.h"
#include "
href../../../../common/math/credit_cds/numint_h_src.pdfnumint.h"

extern "C" {
```

```

#include "pnl/pnl_random.h"
}

using namespace std;

// CIR++ Short Rate
// Piecewise Constant Interpolation
class CIRppSR
{
public:
    CIRppSR(double k = 0, double theta = 0, double sigma = 0, double x0 = 0,
            double T = 0,
            string inputFileName = "",
            double precision = 0.001);
    CIRppSR(double k, double theta, double sigma, double x0,
            double T,
            vector<double> &zMat,
            vector<double> &zRates,
            double precision = 0.001);

    virtual ~CIRppSR()
    {
        delete []_arrayPhi;
        delete []_arrayIntegralsPhi;
        delete []_arrayExpMinusIntegralsPhi;
    }

    double MarketZC(double t) const;
    double Compute_ZC_CIR(double t) const;
    double Compute_ZC_NI(double t) const;
    double Phi(double t) const; // the shift

    typedef double (CIRppSR::*PtrFunction)(double) const;
    double NumericalIntegration_ofPhi_SS(double t) const;
    double GetIntegral_ofPhi(double t) const;

    // for Monte-Carlo purposes: set the current point of the diffusion
    // to the start point

```

```

void Restart()
{
    _xi = _x0;
    _indexOf_xi = -1;
}

double Get_k() const
{
    return _k;
}
double Get_theta() const
{
    return _theta;
}
double Get_sigma() const
{
    return _sigma;
}
double Get_x0() const
{
    return _x0;
}
double Get_xi() const
{
    return _xi;
}
double Get_T() const
{
    return _T;
}
int Get_N() const
{
    return __N;
}
void SetPrecision(double precision);
virtual void Set_T(double T);
double GetStep() const
{
    return _precision;
}

```

```

    void Write(string filename) const;

protected:
    double _k;
    double _theta;
    double _sigma;
    double _x0;
    double _xi;
    int _indexOf_xi;
    double _T;
    double _precision;
    int __N;
    int _noIntegrals;
    double _integrationStep;
    string _inputFileName;
    vector<DateRate> _curveZC;
    vector<DateRate> _pConstShortRate;
    double *_arrayIntegralsPhi;
    double *_arrayExpMinusIntegralsPhi;
    double *_arrayPhi;

private:

    void VerifyParameters();
    void ReadData(string fileName);
    void ReadData(vector<double> &zscMat, vector<double> &zscRates);
    void ComputePConstShortRate();
    double IntegralPConst(double t) const;
    double f0_t(double t) const;
    void Fill_arrayPhi();
    void Fill_arrayIntegralsPhi();

    double NumericalIntegration_S(PtrFunction f, double a, double b) const;
};

// CIR++ Default Intensity
// Piecewise Linear Interpolation
//
// CIR++ process  $r(t) = x \backslash \beta(t) + \backslash \phi(t; \backslash \beta)$ ,
// where  $\backslash \beta = (k, \backslash \theta, \backslash \sigma, x_0)$ ,  $\backslash \phi()$  and  $x$  being
// the corresponding shift function and CIR process:

```

```

// dx(t) = k(\ theta - x(t))dt + \ sigma*sqrt(x(t))*dW(t)
// k = speed of mean reversion
// \ theta = long-run mean
// \ sigma = volatility
// Conditions: k, \ theta, \ sigma > 0, 2k*\ theta >= SQR(\ sigma)
class CIRppDI
{
public:
    CIRppDI(double k = 0, double theta = 0, double sigma = 0, double x0 = 1., double T,
            string inputFileName = "",
            double precision = 0.001);
    CIRppDI(double k, double theta, double sigma, double x0, double T,
            vector<double> &spreadMat,
            vector<double> &spreadRates,
            double precision = 0.001);
    virtual ~CIRppDI()
    {
        delete []_arrayPhi;
        delete []_arrayIntegralsPhi;
    }

    double MarketZC(double t) const
    {
        return exp(-IntegralPLin(t));
    }

    double Compute_ZC_CIR(double t) const;
    double Compute_ZC_NI(double t) const;
    double PLinShortRate(double t) const;
    double Phi(double t) const;

    typedef double (CIRppDI::*PtrFunction)(double) const;
    double NumericalIntegration_ofPhi_SS(double t) const;
    double GetIntegral_ofPhi(double t) const;

    void Restart()
    {
        _xi = _x0;
        _indexOf_xi = -1;
    }

    double Get_k() const

```

```

{
    return _k;
}
double Get_theta() const
{
    return _theta;
}
double Get_sigma() const
{
    return _sigma;
}
double Get_x0() const
{
    return _x0;
}
double Get_xi() const
{
    return _xi;
}
double Get_T() const
{
    return _T;
}
int Get_N() const
{
    return __N;
}
double GetPrecision() const
{
    return _precision;
}
void SetPrecision(double precision);
virtual void Set_T(double T);
double GetStep() const
{
    return _precision;
}

void Write(string filename)  const;

protected:

```

```

double _k;
double _theta;
double _sigma;
double _x0;
double _xi;
int _indexOf_xi;
double _T;
double _precision;
int __N;
int _noIntegrals;
double _integrationStep;
string _inputFileName;
vector<DateRate> _pLinShortRate;
double *_arrayIntegralsPhi;
double *_arrayPhi;

private:
    void VerifyParameters();
    void ReadData(string fileName);
    void ReadData(vector<double> &spreadMat, vector<double> &spreadRates);
    double IntegralPLin(double t) const;
    void Fill_arrayPhi();
    void Fill_arrayIntegralsPhi();
    double NumericalIntegration_S(PtrFunction f, double a, double b) const;
};

// Implements the Explicit(0) scheme for the CIR++ Dflt Intensity
class CIRppDI_Explicit0: public CIRppDI
{
public:
    CIRppDI_Explicit0(
        int generator,
        double k = 0, double theta = 0, double sigma = 0, double x0 = 1.,
        double T = 0,
        string inputFileName = "",
        double precision = 0.001);
    CIRppDI_Explicit0(
        int generator,
        double k, double theta, double sigma, double x0, double T,
        vector<double> &spreadMat,
        vector<double> &spreadRates,

```

```

    double precision = 0.001);
virtual ~CIRppDI_Explicit0() {}

virtual double Next()
{
    double brownianIncrement = _sqrt_T_on_N * pnl_rand_normal(_generator);
    return NextI(brownianIncrement);
}
virtual double Next(double &brownianIncrement)
{
    brownianIncrement = _sqrt_T_on_N * pnl_rand_normal(_generator);
    return NextI(brownianIncrement);
}
double ZeroCoupon_MC(double t, int noSim); //ZC price by Monte-Carlo

double ComputeSup(double t, int noSim);
void Set_T(double T);

friend class DefaultTimeCIRpp;

protected:
//    NEWRAN::Normal _normal_rv;
    int _generator;
    double NextI(double increment);

private:
    double _sqrt_T_on_N;
    double _the_same;
    double _lastTerm;
    void SetTerms();
};

// Implements the Explicit(0) scheme for the CIR++ Short Rate
class CIRppSR_Explicit0: public CIRppSR
{
public:
    CIRppSR_Explicit0(
        int generator,
        double k = 0, double theta = 0, double sigma = 0, double x0 = 1.,
        double T = 0,

```

```

        string inputFileName = "",
        double precision = 0.001);
CIRppSR_Explicit0(
    int generator,
    double k, double theta, double sigma, double x0, double T,
    vector<double> &zcmat,
    vector<double> &zcrates,
    double precision = 0.001);
virtual ~CIRppSR_Explicit0() {}

virtual double Next()
{
    double brownianIncrement = _sqrt_T_on_N * pnl_rand_normal(_generator);
    return NextI(brownianIncrement);
}

double ZeroCoupon_MC(double t, int noSim); //ZC price by Monte-Carlo
void Set_T(double T);

protected:

    //NEWTRAN::Normal _normal_rv;
    int _generator;
    double NextI(double increment);
    double _sqrt_T_on_N;

private:

    double _the_same;
    double _lastTerm;
    void SetTerms();
};

class CIRppSR_Explicit0_Correlated: public CIRppSR_Explicit0
{
public:
    CIRppSR_Explicit0_Correlated(
        int generator,
        double k = 0, double theta = 0, double sigma = 0,

```

```

        double x0 = 1.,
        double T = 0, double rho = 0.5,
        string inputFileName = "",
        double precision = 0.001);

CIRppSR_Explicit0_Correlated(
    int generator,
    double k, double theta, double sigma,
    double x0,
    double T, double rho,
    vector<double> &zMat,
    vector<double> &zRates,
    double precision = 0.001);

double Next();
double Next(double brownianIncr1);

double GetRho()
{
    return _rho;
}
private:
    double _rho;
    double _rho_c;
};

class CIRppDI_Explicit0_Correlated: public CIRppDI_Explicit0
{
public:
    CIRppDI_Explicit0_Correlated(int generator, double k = 0, double theta = 0, double sigma = 0,
        double x0 = 1.,
        double T = 0, double rho = 0.5,
        string inputFileName = "",
        double precision = 0.001):
        CIRppDI_Explicit0(generator, k, theta, sigma, x0, T, inputFileName, precision,
            _rho(rho))
    {}
    double Next();
private:
    double _rho;
};

```

```

// Default Time based on a CIR++ process
class DefaultTimeCIRpp
{
public:
    DefaultTimeCIRpp(int generator, double k, double theta, double sigma, double x0,
                     double T, double barrier, string inputFileName,
                     double precision):
        _intensity(generator, k, theta, sigma, x0, T, inputFileName, precision),
        _barrier(barrier),
        _noCancellations(0)
    {}

    DefaultTimeCIRpp(int generator, double k, double theta, double sigma, double x0,
                     double T, double barrier, vector<double> &spreadMat, vector<double> &spreadRates,
                     double precision):
        _intensity(generator, k, theta, sigma, x0, T, spreadMat, spreadRates, precision),
        _barrier(barrier),
        _noCancellations(0)
    {}

    double Next();
    double Next(double *arrayIncrements);

    double SurvivalProb_Market(double t) //Market Survival Probability
    {
        return _intensity.MarketZC(t);
    }

    //Survival Probability by Monte-Carlo
    double SurvivalProb_MC(double t, int noSim);
    double SurvivalProb_CF(double t); // Survi Probability-Closed Form

    double GetPrecision()
    {
        return _intensity.GetPrecision();
    }
    double GetBarrier()
    {
        return _barrier;
    }
}

```

```

    }
    void BarrierParameters()
    {
        std::cout << "Barrier: " << _barrier
                    << ", _noCancellations: " << _noCancellations << endl;
    }
    void Set_T(double T)
    {
        _intensity.Set_T(T);
    }
    int Get_N()
    {
        return _intensity.Get_N();
    }

protected:
    CIRppDI_Explicit0 _intensity;
    double _barrier;
    int _noCancellations;
};

#endif // cirpp.h

#endif //PremiaCurrentVersion

```