

[Help](#)

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

/// \ file cirpp.cpp
/// \ 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

#include <iostream>
#include <cstring>
#include "
href../../../../common/math/credit_cds/cdscirpp_h_src.pdfcirpp.h"

using namespace std;

static void Fatal_err(const char text[100])
{
    char string[100];
    strcpy(string, "*** Error: ");
    strcat(string, text);
    throw logic_error(string);
}

CIRppSR::CIRppSR(double k, double theta, double sigma, double x0, double T,
                 string inputFileName,
                 double precision):
    _k(k), _theta(theta), _sigma(sigma), _x0(x0), _xi(x0), _T(T),
    _precision(precision),
    _inputFileName(inputFileName)
{
    VerifyParameters();
    ReadData(_inputFileName);
    ComputePConstShortRate();
}
```

```

_indexOf_xi = -1;
__N = (int)ceil(_T / _precision);

try
{
    _arrayPhi = new double[__N];
}
catch (bad_alloc)
{
    cerr << "Out of memory!\ n";
    exit(1);
}

Fill_arrayPhi();

_integrationStep = RIEMANN_NUM_INTEGR_PRECISION;
_noIntegrals = (int) floor(_T / _integrationStep);
try
{
    _arrayIntegralsPhi = new double[_noIntegrals];
    _arrayExpMinusIntegralsPhi = new double[_noIntegrals];
}
catch (bad_alloc)
{
    cerr << "Out of memory!\ n";
    exit(1);
}

Fill_arrayIntegralsPhi();
}

CIRppSR::CIRppSR(double k, double theta, double sigma, double x0,
                double T,
                vector<double> &zMat,
                vector<double> &zRates,
                double precision):
    _k(k), _theta(theta), _sigma(sigma), _x0(x0), _xi(x0), _T(T),
    _precision(precision)
{

```

```

VerifyParameters();
ReadData(zcMat, zcRates);
ComputePConstShortRate();

_indexOf_xi = -1;
__N = (int)ceil(_T / _precision);

try
{
    _arrayPhi = new double[__N];
}
catch (bad_alloc)
{
    cerr << "Out of memory!\ n";
    exit(1);
}

Fill_arrayPhi();

_integrationStep = RIEMANN_NUM_INTEGR_PRECISION;
_noIntegrals = (int) floor(_T / _integrationStep);
try
{
    _arrayIntegralsPhi = new double[_noIntegrals];
    _arrayExpMinusIntegralsPhi = new double[_noIntegrals];
}
catch (bad_alloc)
{
    cerr << "Out of memory!\ n";
    exit(1);
}

Fill_arrayIntegralsPhi();
}

void CIRppSR::Fill_arrayIntegralsPhi()
{
    double xi = 0;
    double sum = 0;
    for (int i = 1; i < _noIntegrals + 1; i++)

```

```

    {
        sum += Phi(xi + _integrationStep) * _integrationStep;
        _arrayIntegralsPhi[i - 1] = sum;
        _arrayExpMinusIntegralsPhi[i - 1] = exp(-sum);
        xi += _integrationStep;
    }
}

void CIRppDI::Fill_arrayIntegralsPhi()
{
    double xi = 0;
    double sum = 0;
    for (int i = 1; i < _noIntegrals + 1; i++)
    {
        sum += Phi(xi + _integrationStep) * _integrationStep;
        _arrayIntegralsPhi[i - 1] = sum;
        xi += _integrationStep;
    }
}

double CIRppSR::Phi(double t) const
{
    double _f0_t = f0_t(t);
    double h = sqrt(SQR(_k) + 2 * SQR(_sigma));
    double exp_th = exp(t * h);
    double quot1 = ((_k * _theta) * (exp_th - 1)) / (h + 0.5 * (_k + h) * (exp_th - 1));
    double quot2 = (SQR(h) * exp_th) / SQR(h + 0.5 * (_k + h) * (exp_th - 1));

    return _f0_t - quot1 - _x0 * quot2;
}

double CIRppSR::f0_t(double t) const
{
    int i = 1;
    int _dim = _curveZC.size();
    while ((t > _pConstShortRate[i - 1].date) && (i < _dim))
        i++;

    if (i > _dim)
    {
        return 0;
    }
}

```

```

    return _pConstShortRate[i - 1].rate;
}

double CIRppSR::MarketZC(double t) const
{
    return exp(-IntegralPConst(t));
}

double CIRppSR::IntegralPConst(double t) const
{
    int i = 1;
    int _dim = _curveZC.size();
    double sum = 0.0;
    while ((t >= _pConstShortRate[i - 1].date) && (i < _dim))
    {
        sum +=
            _pConstShortRate[i - 1].rate * (_curveZC[i].date - _curveZC[i - 1].date);
        i++;
    }

    if (i > _dim)
    {
        return 0;
    }

    if (t == _pConstShortRate[i].date)
        return sum;

    sum += _pConstShortRate[i - 1].rate * (t - _curveZC[i - 1].date);

    return sum;
}

void CIRppSR::ComputePConstShortRate()
{
    int _dim = _curveZC.size();
    if (_dim < 2)
    {
        throw logic_error("Insufficient data!");
    }
}

```

```

double r1 = -log(_curveZC[1].rate / _curveZC[0].rate) /
            (_curveZC[1].date - _curveZC[0].date);
DateRate dr0(_curveZC[1].date, r1);
_pConstShortRate.push_back(dr0);

for (int i = 2; i < _dim; i++)
{
    double P_Tim1 = _curveZC[i - 1].rate;
    double Tim1 = _curveZC[i - 1].date;
    double P_Ti = _curveZC[i].rate;
    double Ti = _curveZC[i].date;
    double r_i = (-log(P_Ti / P_Tim1)) / (Ti - Tim1);

    DateRate dr(Ti, r_i);
    _pConstShortRate.push_back(dr);
}

//Numerical Integration Simpson Method
double CIRppDI::NumericalIntegration_S(PtrFunction f, double a, double b) const
{
    if (a == b)
        return 0.;
    if (a > b)
        return - NumericalIntegration_S(f, b, a);

    // begin Even-Test
    if (SIMPSON_NO % 2 != 0)
    {
        cout << "Error: in CDS_NoCorr_MarketData::NumericalIntegratio, "
              << "SIMPSON_NO must be even. Exit." << endl;
        exit(1);
    }

    double h = (b - a) / SIMPSON_NO;

    double xi0 = (this->*f)(a) + (this->*f)(b), xi1 = 0., xi2 = 0.;

    int i;

```

```

//for(i=1; i<=(n-1); i++)
for (i = 1; i <= (SIMPSON_NO - 1); i++)
{
    double x = a + i * h;
    if (i % 2 == 0)
    {
        xi2 += (this->*f)(x);
    }
    else
    {
        xi1 += (this->*f)(x);
    }
}

return h * (xi0 + 2 * xi2 + 4 * xi1) / 3.;
}

```

```

void CIRppSR::ReadData(string fileName)
{
    ifstream in(fileName.c_str());
    if (!in)
    {
        cout << "CIRppSR::ReadData(string fileName): I Error! \ n";
        exit(1);//ok
    }
    //ifstream in(fileName.c_str());
    if (in.eof())
    {
        cout << "CIRppSR::ReadData(string fileName): No data in input file! \ n";
        exit(1);
    }

    {
        double date, price;
        in >> date >> price;
        DateRate dp(date, price);
        _curveZC.push_back(dp);
    }
}

```

```

}
while (!in.eof())
{
    double date, price;
    in >> date >> price;

    double anteriorDate = _curveZC[_curveZC.size() - 1].date;
    if (date <= anteriorDate)
    {
        cout << "*** Error: Market zero-coupon curve is corrupted!\n n";
        exit(1);
    }

    DateRate dp(date, price);
    _curveZC.push_back(dp);
}
ofstream mat("zc3mat.txt"), rat("zc3rat.txt");
for (int j = 0; j < (int)_curveZC.size(); j++)
{
    mat << _curveZC[j].date << endl;
    rat << _curveZC[j].rate << endl;
}
}

void CIRppSR::ReadData(vector<double> &zMat, vector<double> &zRates)
{
    if (zMat.size() != zRates.size())
    {
        throw logic_error("*** Error: CIRppSR: zcMat and zcRates arrays have not t");
    }

    DateRate dp(zMat[0], zRates[0]);
    _curveZC.push_back(dp);

    for (int i = 1; i < (int)zMat.size(); i++)
    {
        if (zMat[i] <= zMat[i - 1])
        {
            throw logic_error("*** Error: CIRppSR: Market zero-coupon curve is cor");
        }
    }
}

```



```

        DateRate dp(zcMat[i], zcRates[i]);
        _curveZC.push_back(dp);
    }
}

void CIRppDI::ReadData(string fileName)
{
    ifstream input(fileName.c_str());
    if (!input)
    {
        string s("I Error: no file named ");
        s = s + fileName.c_str();
        Fatal_err(s.c_str());
    }
    ifstream in(fileName.c_str());
    if (in.eof())
    {
        string s("I Error: no data in input file named ");
        s = s + fileName.c_str();
        Fatal_err(s.c_str());
    }

    {
        double date, price;
        in >> date >> price;
        DateRate dp(date, price);
        _pLinShortRate.push_back(dp);
    }
    while (!in.eof())
    {
        double date, price;
        in >> date >> price;

        double anteriorDate = _pLinShortRate[_pLinShortRate.size() - 1].date;
        if (date <= anteriorDate)
        {
            cout << fileName.c_str() << ": aici: " << date << " "
                 << anteriorDate << endl;
            Fatal_err("*** Error: Market zero-coupon curve is corrupted!");
        }
    }
}

```

```

    }

    DateRate dp(date, price);
    _pLinShortRate.push_back(dp);
}

void CIRppDI::ReadData(vector<double> &spreadMat, vector<double> &spreadRates)
{
    if (spreadMat.size() != spreadRates.size())
    {
        throw logic_error("*** Error: CIRppDI: spreadMat and spreadRates arrays ha
    }

    DateRate dp(spreadMat[0], spreadRates[0]);
    _pLinShortRate.push_back(dp);

    for (int i = 1; i < (int)spreadMat.size(); i++)
    {
        if (spreadMat[i] <= spreadMat[i - 1])
        {
            throw logic_error("*** Error: CIRppDI: Market cerdit curve curve is co
        }
        DateRate dp(spreadMat[i], spreadRates[i]);
        _pLinShortRate.push_back(dp);
    }
}

void CIRppDI::VerifyParameters()
{
    if ((2 * _k * _theta) < SQR(_sigma))
    {
        Fatal_err("Parameters of CIR process do not satisfy initial condition: 2*_
    }

    if (_x0 <= 0)
    {

```

```

        Fatal_err("Starting point for CIR process must be strictly positive.");
    }

}

void CIRppSR::VerifyParameters()
{
    if ((2 * _k * _theta) < SQR(_sigma))
        Fatal_err("Parameters of CIR process do not satisfy initial conditions.");
    if (_x0 <= 0)
        Fatal_err("Starting point for CIR process must be strictly positive.");
}

// compute _pLinShortRate(t),
// return -1 as error code if t is not in the domain of
// the function _pLinShortRate(.)
double CIRppDI::PLinShortRate(double t) const
{
    int dim = _pLinShortRate.size();

    double x1 = _pLinShortRate[0].date;
    double y1 = _pLinShortRate[0].rate;

    if (t < x1)
    {
        return -1; // error
    }

    int i = 1;
    while ((t > _pLinShortRate[i].date) && (i < dim))
    {
        i++;
    }

    if (i == dim)
    {
        return -1; // error
    }
}

```

```

    }

    if (t == _pLinShortRate[i].date)
        return _pLinShortRate[i].rate;

    x1 = _pLinShortRate[i - 1].date;
    y1 = _pLinShortRate[i - 1].rate;
    double x2, y2;
    x2 = _pLinShortRate[i].date;
    y2 = _pLinShortRate[i].rate;
    double a, b;
    a = (y1 - y2) / (x1 - x2);
    b = y1 - x1 * a;

    return a * t + b;
}

//compute  $\int_0^t \text{\_pLinShortRate}(s) ds$ 
double CIRppDI::IntegralPLin(double t) const
{
    int dim = _pLinShortRate.size();

    double x1 = _pLinShortRate[0].date;
    double y1 = _pLinShortRate[0].rate;
    double x2;
    double y2;

    if (t <= x1) return 0.0;

    double a, b;
    double sum = 0.0;
    int i = 1;
    while ((t > _pLinShortRate[i].date) && (i < dim))
    {
        x2 = _pLinShortRate[i].date;
        y2 = _pLinShortRate[i].rate;
        a = (y1 - y2) / (x1 - x2);
        b = y1 - x1 * a;

        sum += (a * (x2 * x2 - x1 * x1)) / 2. + b * (x2 - x1);
    }
}

```

```

        x1 = x2;
        y1 = y2;
        i++;
    }
    if (i == dim) return sum;

    x2 = _pLinShortRate[i].date;
    y2 = _pLinShortRate[i].rate;
    a = (y1 - y2) / (x1 - x2);
    b = y1 - x1 * a;

    sum += (a * (t * t - x1 * x1)) / 2. + b * (t - x1);
    return sum;
}

double CIRppDI::Compute_ZC_NI(double t) const
{
    //cout <<"CIRppDI::Compute_ZC_NI : " << GetIntegral_ofPhi(t) << endl;
    return exp(- GetIntegral_ofPhi(t)) * Compute_ZC_CIR(t);
}

double CIRppSR::Compute_ZC_NI(double t) const
{
    //cout <<"CIRppSR::Compute_ZC_NI: " << GetIntegral_ofPhi(t) << endl;
    return exp(- GetIntegral_ofPhi(t)) * Compute_ZC_CIR(t);
}

double CIRppSR::Compute_ZC_CIR(double t) const
{
    // :WARNING:
    // 1. The threshold 0.0001 used here is arbitrary
    // 2. What is the solution when compute ZC(t, T) ?
    if (t < 0.0001)
        return 1;

    double h = sqrt(SQR(_k) + 2 * SQR(_sigma));
    double exp_th = exp(t * h);
    double denominator = h + 0.5 * (_k + h) * (exp_th - 1);
    double A = (h * exp((h + _k) * t * 0.5)) / denominator;
    A = pow(A, (2 * _k * _theta) / SQR(_sigma));

```

```

    double B = (exp_th - 1) / denominator;

    return A * exp(-B * _x0);
}

double CIRppDI::Compute_ZC_CIR(double t) const
{
    // :WARNING:
    // 1. The threshold 0.0001 used here is arbitrary
    // 2. What is the solution when compute ZC(t, T) ?
    if (t < 0.0001)
        return 1;

    double h = sqrt(SQR(_k) + 2 * SQR(_sigma));
    double exp_th = exp(t * h);
    double denominator = h + 0.5 * (_k + h) * (exp_th - 1);
    double A = (h * exp((h + _k) * t * 0.5)) / denominator;
    A = pow(A, (2 * _k * _theta) / SQR(_sigma));
    double B = (exp_th - 1) / denominator;

    return A * exp(-B * _x0);
}

//Simpson Numerical Integration
double CIRppSR::NumericalIntegration_S(PtrFunction f, double a, double b) const
{
    if (a == b)
        return 0.;
    if (a > b)
        return - NumericalIntegration_S(f, b, a);

    // begin Even-Test
    if (SIMPSON_NO % 2 != 0)
    {
        Fatal_err("Error: in CIRppSR::NumericalIntegration_S, SIMPSON_NO must be e
    }

    double h = (b - a) / SIMPSON_NO;

    double xi0 = (this->*f)(a) + (this->*f)(b);

```

```

double xi1 = 0.;
double xi2 = 0.;

int i;
for (i = 1; i <= (SIMPSON_NO - 1); i++)
{
    double x = a + i * h;
    if (i % 2 == 0)
    {
        xi2 += (this->*f)(x);
    }
    else
    {
        xi1 += (this->*f)(x);
    }
}

return h * (xi0 + 2 * xi2 + 4 * xi1) / 3.;
}

//compute Integral_0^t Phi(t) dt
double CIRppDI::NumericalIntegration_ofPhi_SS(double t) const
{
    int dim = _pLinShortRate.size();

    double x1 = _pLinShortRate[0].date;
    double x2;

    if (t <= x1) return 0.0;

    //double a, b;
    double sum = 0.0;
    int i = 1;
    while ((t > _pLinShortRate[i].date) && (i < dim))
    {
        x2 = _pLinShortRate[i].date;
        //cout << x1 << " " << x2 << endl;
        sum += NumericalIntegration_S(&CIRppDI::Phi, x1, x2);
        x1 = x2;
        i++;
    }
}

```

```

    if (i == dim) return sum;

    //cout << x1 << " " << t << endl;
    sum += NumericalIntegration_S(&CIRppDI::Phi, x1, t);

    return sum;
}

//compute Integral_0^t Phi(t) dt
//Secvential riemann Sums method
double CIRppSR::NumericalIntegration_ofPhi_SS(double t) const
{
    int dim = _pConstShortRate.size();

    double x1 = _pConstShortRate[0].date;
    double x2;

    if (t <= x1) return 0.0;

    double sum = NumericalIntegration_S(&CIRppSR::Phi, 0, x1);
    int i = 1;
    while ((t > _pConstShortRate[i].date) && (i < dim))
    {
        x2 = _pConstShortRate[i].date;
        sum += NumericalIntegration_S(&CIRppSR::Phi, x1, x2);
        x1 = x2;
        i++;
    }
    if (i == dim) return sum;

    //cout << x1 << " " << t << endl;
    sum += NumericalIntegration_S(&CIRppSR::Phi, x1, t);

    return sum;
}

// return Integral_0^t Phi(t) dt from precomputed array
double CIRppSR::GetIntegral_ofPhi(double t) const
{
    int i;

```



```

double xi = _integrationStep;
if (t < xi)
    return 0;
for (i = 1; i < _noIntegrals; i++)
{
    if (t <= xi)
        break;
    xi += _integrationStep;
}
if (i > _noIntegrals - 1)
    return _arrayIntegralsPhi[_noIntegrals - 1];
return _arrayIntegralsPhi[i];
}

// return Integral_0^t Phi(t) dt from precomputed array
double CIRppDI::GetIntegral_ofPhi(double t) const
{
    int i;
    double xi = _integrationStep;
    if (t < xi)
        return 0;
    for (i = 0; i < _noIntegrals - 1; i++)
    {
        if (t <= xi)
            break;
        xi += _integrationStep;
    }
    assert(_noIntegrals != 1);
    if (i > _noIntegrals - 1)
        return _arrayIntegralsPhi[_noIntegrals - 1];
    assert(i < _noIntegrals);
    return _arrayIntegralsPhi[i];
}

double CIRppDI::Phi(double t) const
{
    double f0_t = PLinShortRate(t);
    double h = sqrt(SQR(_k) + 2 * SQR(_sigma));

```

```

double exp_th = exp(t * h);
double quot1 = ((_k * _theta) * (exp_th - 1)) / (h + 0.5 * (_k + h) * (exp_th - 1));
double quot2 = (SQR(h) * exp_th) / SQR(h + 0.5 * (_k + h) * (exp_th - 1));

return f0_t - quot1 - _x0 * quot2;
}

void CIRppDI::Fill_arrayPhi()
{
    double ti = _precision; //timeStep;
    for (int j = 0; j < __N; j++)
    {
        _arrayPhi[j] = Phi(ti);
        ti += _precision; //timeStep;
    }
}

void CIRppSR::Fill_arrayPhi()
{
    double ti = _precision; //timeStep;
    for (int j = 0; j < __N; j++)
    {
        _arrayPhi[j] = Phi(ti);
        ti += _precision; //timeStep;
    }
}

CIRppDI::CIRppDI(double k, double theta, double sigma, double x0, double T,
                 string inputFileName,
                 double precision):
_k(k), _theta(theta), _sigma(sigma), _x0(x0), _xi(x0), _T(T),
_precision(precision),
_inputFileName(inputFileName)
{
    VerifyParameters();
    ReadData(_inputFileName);
    _indexOf_xi = -1;
    __N = (int)ceil(_T / _precision);
}

```

```

try
{
    _arrayPhi = new double[_N];
}
catch (bad_alloc)
{
    cerr << "Out of memory!\ n";
    exit(1);
}
Fill_arrayPhi();

_integrationStep = RIEMANN_NUM_INTEGR_PRECISION;
_noIntegrals = (int) floor(_T / _integrationStep);
try
{
    _arrayIntegralsPhi = new double[_noIntegrals];
}
catch (bad_alloc)
{
    cerr << "Out of memory!\ n";
    exit(1);
}
Fill_arrayIntegralsPhi();

}

CIRppDI::CIRppDI(double k, double theta, double sigma, double x0, double T,
                vector<double> &spreadMat,
                vector<double> &spreadRates,
                double precision):
_k(k), _theta(theta), _sigma(sigma), _x0(x0), _xi(x0), _T(T),
_precision(precision)
{
    VerifyParameters();
    ReadData(spreadMat, spreadRates);
    _indexOf_xi = -1;
    _N = (int)ceil(_T / _precision);

    try
    {

```

```

        _arrayPhi = new double[_N];
    }
    catch (bad_alloc)
    {
        cerr << "Out of memory!\n n";
        exit(1);
    }
    Fill_arrayPhi();

    _integrationStep = RIEMANN_NUM_INTEGR_PRECISION;
    _noIntegrals = (int) floor(_T / _integrationStep);
    try
    {
        _arrayIntegralsPhi = new double[_noIntegrals];
    }
    catch (bad_alloc)
    {
        cerr << "Out of memory!\n n";
        exit(1);
    }
    Fill_arrayIntegralsPhi();
}

void CIRppDI::SetPrecision(double precision)
{
    _precision = precision;
    _N = (int)ceil(_T / _precision);

    delete []_arrayPhi;
    try
    {
        _arrayPhi = new double[_N];
    }
    catch (bad_alloc)
    {
        cerr << "Out of memory!\n n";
        exit(1);
    }
    Fill_arrayPhi();
}

void CIRppSR::SetPrecision(double precision)

```

```

{
    _precision = precision;
    __N = (int)ceil(_T / _precision);
    delete []_arrayPhi;
    try
    {
        _arrayPhi = new double[__N];
    }
    catch (bad_alloc)
    {
        cerr << "Out of memory!\ n";
        exit(1);
    }

    Fill_arrayPhi();
}

void CIRppDI::Set_T(double T)
{
    _T = T;
    __N = (int)ceil(_T / _precision);
}

void CIRppSR::Set_T(double T)
{
    _T = T;
    __N = (int)ceil(_T / _precision);
}

void CIRppDI::Write(string filename) const
{
    ofstream output(filename.c_str());

    output << "CIRppDI process {x_t, t>=0}. The parameters of x:\ n k = " << _k
        << "\ n theta = " << _theta << "\ n sigma = " << _sigma << "\ n x_0 = "
        << _x0 << "\ n T = " << _T << "\ n precision = "
        << _precision << "\ n N = " << __N << "\ n";
}

```

```

CIRppSR_Explicit0::
CIRppSR_Explicit0(
    int generator,
    double k, double theta, double sigma, double x0,
    double T,
    string inputFileName,
    double precision):
    CIRppSR(k, theta, sigma, x0, T, inputFileName, precision)
    , _generator(generator)
{

    SetTerms();
    //cout << "CIRppSR_Explicit0 Constr\ n";
}

CIRppSR_Explicit0::
CIRppSR_Explicit0(
    int generator,
    double k, double theta, double sigma, double x0, double T,
    vector<double> &zMat,
    vector<double> &zRates,
    double precision):
    CIRppSR(k, theta, sigma, x0, T, zMat, zRates, precision)
    , _generator(generator)
{

    SetTerms();
    //cout << "CIRppSR_Explicit0 Constr\ n";
}

CIRppDI_Explicit0::
CIRppDI_Explicit0(
    int generator,
    double k, double theta, double sigma, double x0,
    double T,
    string inputFileName,
    double precision):
    CIRppDI(k, theta, sigma, x0, T, inputFileName, precision)
    , _generator(generator)
{

```

```

    SetTerms();
    //cout << "CIRppDI_Explicit0 Constr\ n";
}

CIRppDI_Explicit0::
CIRppDI_Explicit0(
    int generator,
    double k, double theta, double sigma, double x0, double T,
    vector<double> &spreadMat,
    vector<double> &spreadRates,
    double precision):
    CIRppDI(k, theta, sigma, x0, T, spreadMat, spreadRates, precision),
    _generator(generator)
{

    SetTerms();
    //cout << "CIRppDI_Explicit0 Constr\ n";
}

CIRppSR_Explicit0_Correlated::
CIRppSR_Explicit0_Correlated(
    int generator,
    double k, double theta, double sigma, double x0,
    double T, double rho,
    string inputFileName,
    double precision):
    CIRppSR_Explicit0(generator, k, theta, sigma, x0, T, inputFileName, precision)
    _rho(rho)
{
    //cout << "CIRppSR_Explicit0_Correlated Constr\ n";

    if ((_rho < -1) || (_rho > 1))
        throw logic_error("Correlation must be in [-1, 1].\ n");

    _rho_c = sqrt(1 - SQR(_rho));
}

CIRppSR_Explicit0_Correlated::
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):
CIRppSR_Explicit0(generator, k, theta, sigma, x0, T, zMat, zRates, precision,
_rho(rho)
{
    //cout << "CIRppSR_Explicit0_Correlated Constr\ n";

    if ((_rho < -1) || (_rho > 1))
        throw logic_error("Correlation must be in [-1, 1].\ n");

    _rho_c = sqrt(1 - SQR(_rho));
}
void CIRppSR_Explicit0::SetTerms()
{
    _sqrt_T_on_N = sqrt(_T / __N);
    _the_same = 1 - (_k * _T) / (2 * __N);
    _lastTerm = (_k * _theta - SQR(_sigma) / 4) * (_T / __N);
}

void CIRppDI_Explicit0::SetTerms()
{
    _sqrt_T_on_N = sqrt(_precision);
    _the_same = 1 - _precision * _k / 2;
    _lastTerm = (_k * _theta - SQR(_sigma) / 4) * _precision;
}

void CIRppDI_Explicit0::Set_T(double T)
{
    CIRppDI::Set_T(T);
    //SetTerms();
}

void CIRppSR_Explicit0::Set_T(double T)
{
    CIRppSR::Set_T(T);
    SetTerms();
}

```



```

double CIRppSR_Explicit0::NextI(double increment)
{
    //cout << "CIRppSR_Explicit0::NextI\ n";
    //Explicit(0) scheme, alfonsi, "on the simulation of the cir process"
    double firstTermInSQR = _the_same * sqrt(_xi);
    double secondTermInSQR = (_sigma * increment) / (2 * _the_same);
    double x_i_plus_1 = SQR(firstTermInSQR + secondTermInSQR) + _lastTerm;

    _xi = x_i_plus_1;
    _indexOf_xi++;
    assert(_indexOf_xi < __N);
    return x_i_plus_1 + _arrayPhi[_indexOf_xi];
}

```

```

double CIRppDI_Explicit0::NextI(double increment)
{

    //Explicit(0) scheme, alfonsi, "on the simulation of the cir process"
    double firstTermInSQR = _the_same * sqrt(_xi);
    double secondTermInSQR = (_sigma * increment) / (2 * _the_same);
    double x_i_plus_1 = SQR(firstTermInSQR + secondTermInSQR) + _lastTerm;

    _xi = x_i_plus_1;
    //cout << _xi << " ";
    _indexOf_xi++;
    assert(_indexOf_xi < __N);
    return x_i_plus_1 + _arrayPhi[_indexOf_xi];
}

```

```

//ZC price by Monte-Carlo
double CIRppDI_Explicit0::ZeroCoupon_MC(double t, int noSim)
{

    Set_T(t);

    double timeStep = GetStep();
    int noCIRpp_SimPoints = Get_N();
    double sum = 0.;

```

```

//double sum_int = 0.;
for (int i = 0; i < noSim; i++)
{
    Restart();

    double sum_int = 0.;
    //sum_int = 0.;
    for (int j = 0; j < noCIRpp_SimPoints; j++)
        sum_int += Next();

    sum += exp(-sum_int * timeStep);
}

return sum / noSim;
}
//ZC price by Monte-Carlo
double CIRppSR_Explicit0::ZeroCoupon_MC(double t, int noSim)
{
    Set_T(t);

    double timeStep = GetStep();
    int noCIRpp_SimPoints = Get_N();
    double sum = 0.;
    for (int i = 0; i < noSim; i++)
    {
        Restart();

        double sum_int = 0.;
        //sum_int = 0.;
        for (int j = 0; j < noCIRpp_SimPoints; j++)
            sum_int += Next();

        sum += exp(-sum_int * timeStep);
    }

    return sum / noSim;
}

double CIRppDI_Explicit0::ComputeSup(double t, int noSim)
{

```

```

double sup = 0;
Set_T(t);
int noCIRpp_SimPoints = Get_N();
for (int j = 0; j < noSim; j++)
{
    Restart();
    for (int i = 0; i < noCIRpp_SimPoints; i++)
    {
        double cirpp = Next();
        if (cirpp > sup)
            sup = cirpp;
    }
}

return sup;
}

double CIRppDI_Explicit0_Correlated::Next()
{
    double brownianIncr1 = sqrt(_T / __N) * pnl_rand_normal(_generator);
    double brownianIncr2 = sqrt(_T / __N) * pnl_rand_normal(_generator);
    double corrIncr = _rho * brownianIncr1 + sqrt(1 - SQR(_rho)) * brownianIncr2;

    return NextI(corrIncr);
}

double CIRppSR_Explicit0_Correlated::Next()
{
    double brownianIncr1 = _sqrt_T_on_N * pnl_rand_normal(_generator);
    double brownianIncr2 = _sqrt_T_on_N * pnl_rand_normal(_generator);
    double corrIncr = _rho * brownianIncr1 + _rho_c * brownianIncr2;

    return NextI(corrIncr);
}

double CIRppSR_Explicit0_Correlated::Next(double brownianIncr1)
{
    double brownianIncr2 = _sqrt_T_on_N * pnl_rand_normal(_generator);
    double corrIncr = _rho * brownianIncr1 + _rho_c * brownianIncr2;

```

```

    return NextI(corrIncr);
}

// Direct simulation of the time of default
// pg. 216 from P.J. Schonbucher, "Credit derivative pricing models"
double DefaultTimeCIRpp::Next()
{
    // NEWRAN::Uniform triggerLevel;
    double U = pnl_rand_uni(_intensity._generator);

    //cout << "U=" << U << " ";
    double lnU = log(U);
    double value_of_TheIntegral = _intensity.Get_T() * _barrier;
    if (lnU < -value_of_TheIntegral) //_noCancellations++;
    {
        return 0;
    }

    double lnDCP = 0.; //log of Default Countdown Process
    double defaultTime = 0.;
    double timeStep = _intensity.GetStep();
    int noCIRpp_SimPoints = _intensity.Get_N();
    //cout << "noCIRpp_SimPoints = " << noCIRpp_SimPoints << endl;

    _intensity.Restart();
    int indexTimeInterval = 0;

    for (indexTimeInterval = 0; indexTimeInterval < noCIRpp_SimPoints;
        indexTimeInterval++)
    {
        double cirpp = _intensity.Next();
        lnDCP -= cirpp * timeStep;
        defaultTime += timeStep;
        if (lnU > lnDCP)
            return defaultTime;
        //return indexTimeInterval * timeStep;
    }
}

```

```

    return 0;
}

// Direct simulation of the time of default
// pg. 216 from P.J. Schonbucher, "Credit derivative pricing models"
double DefaultTimeCIRpp::Next(double *arrayIncrements)
{
    //NEWTRAN::Uniform triggerLevel;
    double U = pnl_rand_uni(_intensity._generator);

    double lnU = log(U);
    double value_of_TheIntegral = _intensity.Get_T() * _barrier;
    if (lnU < -value_of_TheIntegral) //_noCancellations++;
    {
        return 0;
    }

    double lnDCP = 0.;//log of Default Countdown Process
    double defaultTime = 0.;
    double timeStep = _intensity.GetStep();
    int noCIRpp_SimPoints = _intensity.Get_N();

    _intensity.Restart();
    int indexTimeInterval = 0;

    for (indexTimeInterval = 0; indexTimeInterval < noCIRpp_SimPoints;
        indexTimeInterval++)
    {
        double cirpp = _intensity.Next(arrayIncrements[indexTimeInterval]);
        lnDCP -= cirpp * timeStep;
        defaultTime += timeStep;
        if (lnU > lnDCP)
            return defaultTime;
    }

    return 0;
}

```

```

double DefaultTimeCIRpp::SurvivalProb_MC(double t, int noSim)
{
    double sum = 0.;

    _intensity.Set_T(t);
    for (int i = 0; i < noSim; i++)
    {
        double tau = Next();
        if (tau == 0.)
        {
            sum += 1.;
        }
    }

    return sum / noSim;
}

double DefaultTimeCIRpp::SurvivalProb_CF(double t)
{
    return _intensity.MarketZC(t);
}

#endif //PremiaCurrentVersion

```