```
   Help
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion <
     (2007+2) //The "#else" part of the code will be freely av
    ailable after the (year of creation of this file + 2)
#else

/// {file cdscirppmc.cpp
/// {brief CDS_CIRpp_MC class
/// {author  M. Ciuca (MathFi, ENPC)
/// {note (C) Copyright Premia 8 - 2006, under Premia 8 Sof
    tware license
//
//  Use, modification and distribution are subject to the
//  Premia 8 Software license

#include <iostream>
#include "base.h"
#include "cdscirppmc.h"


inline bool EQ(double x, double y)
{
  if(((-TOLERANCE+y)<x) && (x<(TOLERANCE+y)))
    return true;
  return false;
}


inline bool EQ(double x, double y, double tolerance)
{
  if(((-tolerance+y)<=x) && (x<=(tolerance+y)))
    return true;
  return false;
}



CDS_CIRpp_MC::CDS_CIRpp_MC(int generator,
                           double mrIntensity, double thet
    aIntensity,
               double sigmaIntensity, double y0,
```

```
                vector<double>& spreadMat,
                vector<double>& spreadRates,
                double mrRate, double thetaRate,
    double sigmaRate,
                double x0_r,
                vector<double>& RatesMat,
                vector<double>& Rates,
                double correlation, double maturity,
    double period,
                double recovery,
                int Nsim, double precision, double bar
    rier):
_tau( generator, mrIntensity, thetaIntensity, sigma
    Intensity, y0,
   maturity, barrier, spreadMat, spreadRates, precision),
   _shortRate( generator, mrRate, thetaRate, sigmaRate, x0
    _r,
   maturity, correlation, RatesMat, Rates, precision),
   _Z(1-recovery),
   _noTau_Sim(Nsim),
   _b(0.),
   _c(0.)
{
  //std::cout << "nMc : " << _noTau_Sim << endl;

  _timesT.push_back(0.0);
  double t, yearFrac;
  t = yearFrac = (12./period);
  int periodN = static_cast <int> ( maturity / yearFrac );
  for(int i=0; i<periodN; i++)
  {
    _timesT.push_back(t);
    t += yearFrac;
  }
}

void CDS_CIRpp_MC::WriteCharacteristics()
{

  cout << "CDS_CIRpp_MC: {n"
    << "nMC: " << _noTau_Sim << endl
```

```
    << "scheme precision: " << _tau.GetPrecision() << endl
    << "barrier: " <<  _tau.GetBarrier() << endl
    << "correlation: " << _shortRate.GetRho() << endl;

}


double CDS_CIRpp_MC::CdsRate()
{
  double sumI1;
  double sumI2;
  double sumS;
  MonteCarlo(sumI1, sumI2, sumS, _noTau_Sim);

  cout << "DL: " << _Z * sumI2  / _noTau_Sim << endl
    << "PL: " << (sumI1 + sumS) / _noTau_Sim << endl;

  //printf("DL: %lf{n", _Z * sumI2 / _noTau_Sim);

  //cout << "I1, I2, S: " << sumI1 << " " << sumI2 << " "
    << sumS << endl;
  cout << "I1, I2, S: " << sumI1/_noTau_Sim << " " << sumI
    2/_noTau_Sim << " " << sumS/_noTau_Sim << endl;
  //sumI1 /= _noTau_Sim;   sumI2 /= _noTau_Sim;  sumS  /=
    _noTau_Sim;
  return (_Z * sumI2)/(sumI1 + sumS);
}
double CDS_CIRpp_MC::CdsRate(double& DefaultLeg, double&
    PaymentLeg,
               double& std_dev_DefaultLeg, double&     std_dev_PaymentLeg)
{
  MonteCarlo(DefaultLeg, PaymentLeg, std_dev_DefaultLeg,     std_dev_PaymentLeg,
  return DefaultLeg/PaymentLeg;
}

int CDS_CIRpp_MC::MonteCarlo(double& DefaultLeg, double&
    PaymentLeg,
               double& std_dev_DefaultLeg, double&     std_dev_PaymentLeg, int n
{

  double sumI1 = 0.0;
```

```
  double sumS  = 0.0;
  double sumI2 = 0.0;
  double sumI1_sqr = 0.0;
  double sumI2_sqr  = 0.0;
  double sumS_sqr = 0.0;

  bool _reset_T = false;
  for(int n=0; n<nS; n++)
  {
    // TO DO
    Generate_Yi(sumI1, sumI2, sumS, sumI1_sqr, sumI2_sqr,
     sumS_sqr,_reset_T);
  }

  DefaultLeg = _Z * sumI2 / nS;
  PaymentLeg = (sumI1 + sumS) / nS;

  //sqrt( (sumOfSqrDefaultLegs - nMC*SQR(price))/(nMC - 1)
     );
  std_dev_DefaultLeg = sqrt( (sumI2_sqr - nS*SQR(sumI2 /
    nS))/(nS - 1) );
  std_dev_PaymentLeg = sqrt( (sumI1_sqr+sumS_sqr - nS*SQR(
    PaymentLeg))/(nS - 1) );

  return 1;
}
int CDS_CIRpp_MC::MonteCarlo(double& sumI1, double& sumI2,
    double& sumS, int nS)
{

  sumI1 = 0.0;
  sumS  = 0.0;
  sumI2 = 0.0;
  //cout << "I1, I2, S: " << sumI1 << " " << sumI2 << " "
    << sumS << endl;
  //ofstream out("cirex_debTXT.cpp");
  bool _reset_T = false;
  for(int n=0; n<nS; n++)
  {
    Generate_Yi(sumI1, sumI2, sumS, _reset_T);
  }
```

```
  sumI1 /= nS;
  sumI2 /= nS;
  sumS  /= nS;

  //cout << "i1, i2, s: " << sumI1 << " " << sumI2 << " "
    << sumS << endl;
  return 1;
}


void CDS_CIRpp_MC::Estimate_b_and_c(double meanI1, double
    meanI2, double meanS, int N)
{

  double meanI = meanI2, meanJ = meanI1+meanS;
  double _T = _timesT[_timesT.size() - 1];

  double defaultProbability = 1.0 - _tau.SurvivalProb_Mar
    ket(_T);

  double numerator_b = 0.0, numerator_c = 0.0;
  double denominator = 0.0;
  double sumI=0., sumJ=0.;
  bool _reset_T = false;
  for(int n=0; n<N; n++)
  {
    double I1i = 0.0, I2i  = 0.0, Si = 0.0;
    bool _default = Generate_Yi(I1i, I2i, Si, _reset_T);

    double I = I2i;
    double J = I1i + Si;
    sumI += I;
    sumJ += J;

    if(_default)
    {
      numerator_b += (1 - defaultProbability)*(I - mea
    nI);
      numerator_c += (1 - defaultProbability)*(J - meanJ
    );
```

```
      denominator += SQR(1 - defaultProbability);
    }
    else
    {
      numerator_b += defaultProbability * meanI;
      numerator_c += defaultProbability * (meanJ - Si);
      denominator += SQR(defaultProbability);
    }

  }
  _b = numerator_b / denominator;
  _c = numerator_c / denominator;
}



bool CDS_CIRpp_MC::Generate_Yi(double& sumI1, double& sumI2
    , double& sumS,
                  bool& reset_T)
{
  double _T = _timesT[_timesT.size() - 1];
  double tau = _tau.Next();
  //cout << "tau: " << tau << endl;
  int noB;
  double T_betaTau_minus_1=0;
  double timeStep = _shortRate.GetStep();
  if(tau == 0) //the default has not yet occured in T, i.
    e. tau>T
  {
    noB = _timesT.size()-1;
    //cout << n << ", No default, noB: " << noB << endl;
    _shortRate.Set_T(_T);
    if(reset_T)
    {
      _shortRate.Set_T(_T);
      //cout << "T, step:" << _shortRate.GetStep() <<
    endl;
      reset_T = false;
    }

  }
```

```
else //the default has occured before T, i.e. tau<T
{
  //cout << n << " " << tau << endl;
  noB = 1;
  while( tau > _timesT[noB] )
    noB++;
  T_betaTau_minus_1 = _timesT[noB-1];


  if(EQ(tau, _timesT[noB-1], timeStep))
    if(noB > 1) noB--;

  //cout << n << " noB: " << noB << endl;
  //cout << n << " " << tau << ", noB: " << noB << end
  l;

  //set the simulation interval for r: [0, tau]
  _shortRate.Set_T(tau);
  //cout << "tau: " << tau << ", step:" << _shortRate.
  GetStep()<< endl;
  reset_T = true;
}


double *timesB;
try{
  timesB = new double[noB];
} catch(bad_alloc){
  cerr << "Out of memory!{n";
  exit(1);
}


if( noB > 1 )
  for(int _i=0; _i<(noB-1); _i++)
    //see the definition of _timesT in class definition
    timesB[_i] = _timesT[_i+1];
if(tau == 0)
  timesB[noB-1] = _timesT[noB];
else
  timesB[noB-1] = tau;
```

```
_shortRate.Restart();
double *iterative_sums_B;
iterative_sums_B = new double[noB];
double sum_B = 0;


int i = 0;
int noSimSR =  _shortRate.Get_N();
double Ti = timesB[i];



int noSteps_on_Ti = (int)ceil( Ti/timeStep );
int noTi = noB - 1;
for(i=0; i<noTi; i++)
{
  for(int j=0; j<noSteps_on_Ti; j++)
    sum_B += _shortRate.Next();
  iterative_sums_B[i] = sum_B;
}
int noSteps_on_lastInterval = noSimSR - noTi*noSteps_on_
  Ti;
for(int j=0; j<noSteps_on_lastInterval; j++)
  sum_B += _shortRate.Next();
iterative_sums_B[noB-1] = sum_B;



for(i=0; i<noB; i++)
  iterative_sums_B[i] = exp( - timeStep*iterative_sums_
  B[i] );

//ATENTION: I suppose that T1,...,Tn are equispaced !!!
double alpha = _timesT[1] - _timesT[0];

double S = 0;
if(tau==0)
  for(int i=0; i<noB; i++)
    S += iterative_sums_B[i];
else
```

```
    if(noB > 1)
    {
      for(int i=0; i<noB-1; i++)
        S += iterative_sums_B[i];
    }
    S *= alpha;


    if(tau != 0.)
    {
      sumI1 +=  (tau - T_betaTau_minus_1) * iterative_su
    ms_B[noB-1];
      sumI2 += iterative_sums_B[noB-1];
    }

    sumS  += S;

    delete []iterative_sums_B;
    delete []timesB;

    if(tau)
      return true;
    return false;
}


bool CDS_CIRpp_MC::Generate_Yi(double& sumI1, double& sumI2
    , double& sumS,
                  double& sumI1_sqr, double& sumI2_sq
    r, double& sumS_sqr, bool& reset_T)
{
  double _T = _timesT[_timesT.size() - 1];
  double tau = _tau.Next();
  //cout << "tau: " << tau << endl;
  int noB;
  double T_betaTau_minus_1=0;
  double timeStep = _shortRate.GetStep();
  if(tau == 0) //the default has not yet occured in T, i.
    e. tau>T
  {
    noB = _timesT.size()-1;
```

```
  //cout << n << ", No default, noB: " << noB << endl;
  _shortRate.Set_T(_T);
  if(reset_T)
  {
    _shortRate.Set_T(_T);
    //cout << "T, step:" << _shortRate.GetStep() <<
  endl;
    reset_T = false;
  }

}
else //the default has occured before T, i.e. tau<T
{
  //cout << n << " " << tau << endl;
  noB = 1;
  while( tau > _timesT[noB] )
    noB++;
  T_betaTau_minus_1 = _timesT[noB-1];


  if(EQ(tau, _timesT[noB-1], timeStep))
    if(noB > 1) noB--;

  //cout << n << " noB: " << noB << endl;
  //cout << n << " " << tau << ", noB: " << noB << end
  l;

  //set the simulation interval for r: [0, tau]
  _shortRate.Set_T(tau);
  //cout << "tau: " << tau << ", step:" << _shortRate.
  GetStep()<< endl;
  reset_T = true;
}


double *timesB;
try{
  timesB = new double[noB];
} catch(bad_alloc){
  cerr << "Out of memory!{n";
  exit(1);
```

```
}


if( noB > 1 )
  for(int _i=0; _i<(noB-1); _i++)
    //see the definition of _timesT in class definition
    timesB[_i] = _timesT[_i+1];
if(tau == 0)
  timesB[noB-1] = _timesT[noB];
else
  timesB[noB-1] = tau;

_shortRate.Restart();
double *iterative_sums_B;
iterative_sums_B = new double[noB];
double sum_B = 0;


int i = 0;
int noSimSR =  _shortRate.Get_N();
double Ti = timesB[i];



int noSteps_on_Ti = (int)ceil( Ti/timeStep );
int noTi = noB - 1;
for(i=0; i<noTi; i++)
{
  for(int j=0; j<noSteps_on_Ti; j++)
    sum_B += _shortRate.Next();
  iterative_sums_B[i] = sum_B;
}
int noSteps_on_lastInterval = noSimSR - noTi*noSteps_on_
  Ti;
for(int j=0; j<noSteps_on_lastInterval; j++)
  sum_B += _shortRate.Next();
iterative_sums_B[noB-1] = sum_B;



for(i=0; i<noB; i++)
```

```
    iterative_sums_B[i] = exp( - timeStep*iterative_sums_
    B[i] );

  //ATENTION: I suppose that T1,...,Tn are equispaced !!!
  double alpha = _timesT[1] - _timesT[0];

  double S = 0;
  if(tau==0)
    for(int i=0; i<noB; i++)
      S += iterative_sums_B[i];
  else
    if(noB > 1)
    {
      for(int i=0; i<noB-1; i++)
        S += iterative_sums_B[i];
    }
    S *= alpha;


    if(tau != 0.)
    {
      sumI1 +=  (tau - T_betaTau_minus_1) * iterative_su
    ms_B[noB-1];
      sumI2 += iterative_sums_B[noB-1];

      sumI1_sqr += SQR( (tau - T_betaTau_minus_1) *
    iterative_sums_B[noB-1] );
      sumI2_sqr += SQR( iterative_sums_B[noB-1] );
    }

    sumS  += S;
    sumS_sqr += SQR( S );

    delete []iterative_sums_B;
    delete []timesB;

    if(tau)
      return true;
    return false;
}
```

```
double CDS_CIRpp_MC::CdsRate_ControlVariate()
{
  double _T = _timesT[_timesT.size() - 1];
  double defaultProbability = 1.0 - _tau.SurvivalProb_Mar
    ket(_T);

  double sumIi_b = 0.0, sumJi_c = 0.0;
  bool _reset_T = false;
  for(int i=0; i<_noTau_Sim; i++)
  {
    double sumI1i = 0.0, sumI2i  = 0.0, sumSi = 0.0;
    bool _default = Generate_Yi(sumI1i, sumI2i, sumSi, _
    reset_T);

    double Ii = sumI2i;
    double Ji = sumI1i + sumSi;

    if(_default)
    {
      sumIi_b += Ii - _b*(1 - defaultProbability);
      sumJi_c += Ji - _c*(1 - defaultProbability);
    }
    else
    {
      sumIi_b += Ii + _b*defaultProbability;
      sumJi_c += Ji + _c*defaultProbability;
    }
  }
  cout << "DL: " << _Z * sumIi_b / _noTau_Sim << "{nPL: "
    << sumJi_c / _noTau_Sim << endl;
  return (_Z * sumIi_b)/sumJi_c;
}

double CDS_CIRpp_MC::CdsRate_ControlVariate(double& Default
    Leg, double& PaymentLeg,
    double& std_dev_DefaultLeg, double& std_dev_Payment
    Leg)
{
  double _T = _timesT[_timesT.size() - 1];
  double defaultProbability = 1.0 - _tau.SurvivalProb_Mar
    ket(_T);
```

```
double sumIi_b = 0.0, sumJi_c = 0.0, sumIi_sqr_b = 0.0,
  sumJi_sqr_c = 0.0;
bool _reset_T = false;
for(int i=0; i<_noTau_Sim; i++)
{
  double sumI1i = 0.0, sumI2i  = 0.0, sumSi = 0.0;
  bool _default = Generate_Yi(sumI1i, sumI2i, sumSi, _
  reset_T);

  double Ii = sumI2i;
  double Ji = sumI1i + sumSi;

  if(_default)
  {
    sumIi_b += Ii - _b*(1 - defaultProbability);
    sumJi_c += Ji - _c*(1 - defaultProbability);

    sumIi_sqr_b += SQR( Ii - _b*(1 - defaultProbabil
  ity) );
    sumJi_sqr_c += SQR( Ji - _c*(1 - defaultProbabil
  ity) );
  }
  else
  {
    sumIi_b += Ii + _b*defaultProbability;
    sumJi_c += Ji + _c*defaultProbability;

    sumIi_sqr_b += SQR( Ii + _b*defaultProbability );
    sumJi_sqr_c += SQR( Ji + _c*defaultProbability );
  }
}

DefaultLeg = _Z * sumIi_b / _noTau_Sim;
PaymentLeg = sumJi_c / _noTau_Sim;

//sqrt( (sumOfSqrDefaultLegs - nMC*SQR(price))/(nMC - 1)
   );
std_dev_DefaultLeg = sqrt( (sumIi_sqr_b - _noTau_Sim*SQ
  R(sumIi_b / _noTau_Sim))/(_noTau_Sim - 1) );
std_dev_PaymentLeg = sqrt( (sumJi_sqr_c - _noTau_Sim*SQ
```

```
    R(PaymentLeg))/(_noTau_Sim - 1) );


  return (_Z * sumIi_b)/sumJi_c;
}

double CDS_CIRpp_MC::DefaultableZC_MC(double t)
{
  double zc= _shortRate.ZeroCoupon_MC(t, _noTau_Sim);
  double sp= _tau.SurvivalProb_MC(t, _noTau_Sim);
  return zc * sp;
}

double CDS_CIRpp_MC::DefaultableZC_Mkt(double t)
{
  return _shortRate.MarketZC(t) * _tau.SurvivalProb_Marke
    t(t);
}

#endif //PremiaCurrentVersion
```

References