

[Help](#)

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

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

#include "
href../../common/math/credit_cds/intensitycalib_h_src.pdfintensitycalib.h"

#define PIECEWISE_NO PIECEWISE_NUMBER + 1

using namespace std;

typedef double (* PtrFunction)(double, double[][2], double);
typedef double (* PtrF6D)(double, double, double, double, int, double *);
typedef double (* PtrF10bD)(double, double, double[][2], int, double, int, int,
                           double, double, double *);

//*****
//***** PIECEWISE LINEAR INTENSITY *****
//*****

double piecewise_linear(double a, double b, double x)
{
    return a * x + b;
}

int segment(double x1, double y1, double x2, double y2, double *a, double *b)
{
    if (x1 == x2)
        return 1;
}
```

```

    *a = (y1 - y2) / (x1 - x2);
    *b = y1 - x1 * (*a);

    return 0;
}

double pLin_Intensity(double t, double v[][2], int dim)
{
    double x1 = v[0][0];
    double y1 = v[0][1];
    double x2;
    double y2;

    if (t < x1)
    {
        return 0.0;
    }

    double a, b;

    //double sum = 0.0;
    int i = 1;
    while ((t > v[i][0]) && (i < dim))
    {
        i++;
    }

    if (i == dim)
    {
        return 0;
    }

    if (t == v[i][0])
        return v[i][1];

    x1 = v[i - 1][0];

```

```

    y1 = v[i - 1][1];
    x2 = v[i][0];
    y2 = v[i][1];
    segment(x1, y1, x2, y2, &a, &b);

    return a * t + b;
}

```

```

double pLin_Integral(double t, double v[][2], int dim)
{
    double x1 = v[0][0];
    double y1 = v[0][1];
    double x2;
    double y2;

    if (t <= x1) return 0.0;

    double a, b;

    double sum = 0.0;
    int i = 1;
    while ((t > v[i][0]) && (i < dim))
    {
        x2 = v[i][0];
        y2 = v[i][1];
        segment(x1, y1, x2, y2, &a, &b);

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

        x1 = x2;
        y1 = y2;
        i++;
    }

    if (i == dim) return sum;

    x2 = v[i][0];

```

```

    y2 = v[i][1];
    segment(x1, y1, x2, y2, &a, &b);

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

int WriteHazardFunction(double v[][2], int dim, double a, double b, int n,
                        string filename)
{
    ofstream output_data(filename.c_str());
    if (output_data.is_open())
    {
        double t;

        int i;
        for (i = 0; i < n; i++)
        {
            t = a + i * (b - a) / n;
            output_data << t << " " << pLin_Integral(t, v, dim) << endl;
        }

        return 0;
    }

    cout << "0 Error !" << endl;
    exit(1);
    return 1;
}

double DefaultProb(double t, double v[][2], int dim)
{
    return 1 - exp(-pLin_Integral(t, v, dim));
}

int WriteDefaultProb(double v[][2], int dim, double a, double b, int n,
                     string filename)
{
    ofstream output_data(filename.c_str());
    if (output_data.is_open())
    {

```

```

        double t;

        int i;
        for (i = 0; i < n; i++)
        {
            t = a + i * (b - a) / n;
            output_data << t << " " << 1 - exp(-pLin_Integral(t, v, dim))
                        << endl;
        }

        return 0;
    }

    cout << "0 Error !" << endl;
    exit(1);
    return 1;
}

```

```

int Read2DVectorFF(double v[][2], int dim, string filename)
{
    ifstream input_data(filename.c_str());
    if (input_data.is_open())
    {
        int i, j;
        for (i = 0; i < dim; i++)
            for (j = 0; j < 2; j++)
                input_data >> v[i][j];
        return 0;
    }
    cout << "I Error !" << endl;
    exit(1);

    return 1;
}

/*
int Write2DVector(double v[][2], int dim)
{

```

```

    int i;
    for(i=0; i<dim; i++)
    cout << v[i][0] << " " << v[i][1] << endl;

    return 0;

}
*/
int Write2DVectorIF(double v[][2], int dim, string filename)
{

    ofstream output_data(filename.c_str());
    if (output_data.is_open())
    {
        int i;
        for (i = 0; i < dim; i++)
            output_data << v[i][0] << " " << v[i][1] << endl;
        return 0;
    }

    cout << "0 Error !" << endl;
    exit(1);
    return 1;
}

//*****
//*****          CDS PRICING          *****
//*****

double zcb_ciy(double r, double t, double T)
{
    return exp(-r * (T - t));
}

double f2(double r, double piecewiseLinFct[][2], double u)
{
    return zcb_ciy(r, 0, u) * pLin_Intensity(u, piecewiseLinFct, PIECEWISE_NO)
        * exp(-pLin_Integral(u, piecewiseLinFct, PIECEWISE_NO));
}

```

```

double f1(double r, double piecewiseLinFct[][2], double u)
{
    return f2(r, piecewiseLinFct, u) * u;
}

double f_Sum(double r, double piecewiseLinFct[][2], double *timesT, int n0,
             int n)
{
    if (n0 > n)
    {
        throw logic_error("** Error: in the routine f_Sum. Bad input data!");
    }

    double s = 0;

    int i;
    for (i = n0; i <= n; i++)
    {
        s += zcb_ciy(r, 0, timesT[i]) * (timesT[i] - timesT[i - 1])
            * exp(-pLin_Integral(timesT[i], piecewiseLinFct, PIECEWISE_NO));
    }

    return s;
}

//*****
//*****          CDS PRICING          *****
//*****

/*
Composite Simpson's Rule for Numerical Integration
Alg. 4.1, pg 186, from Burden & Faires, "Numerical ananlysis"
Thm. 4.4, pg 186

Compute numerical approximation of  $\int_a^b f(x) dx$ 
Attention: f must be of class  $C^4$  on  $[a, b]$  !!
*/
double numericalIntegration_CompositeSimpson(PtrFunction f,
      double r,
      double piecewiseLinFct[][2],

```

```

        double a, double b)
{
    if (a == b)
        return 0.;
    if (a > b)
        return - numericalIntegration_CompositeSimpson(f, r, piecewiseLinFct, b, a);

    // begin Even-Test
    // to remove later
    //if(n%2 != 0)
    if (SIMPSON_NO % 2 != 0)
    {
        throw logic_error("SIMPSON_NO must be even. Exit.");
    }
    // end Even-Test

    //double h = (b - a)/n;
    double h = (b - a) / SIMPSON_NO;

    double xi0 =
        f(r, piecewiseLinFct, a) + f(r, piecewiseLinFct, 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 += f(r, piecewiseLinFct, x);
        }
        else
        {
            xi1 += f(r, piecewiseLinFct, x);
        }
    }

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

/*

```


Bisection Method for numerically solving one-dimensional equations
 Alg. 2.1, pg 41, from Burden & Faires, "Numerical ananlysis"
 Thm. 2.1, pg 436

Compute a numerical approximation of the solution of equation:
 $f(x) = 0$, x belonging to $[a, b]$.

Attention: f must be continuous on $[a, b]$!!

*/

```
int bisectionPtrF6D(PtrF6D f, double R, double r, double Z, int n,
                  double *timesT, double a, double b, int &maxNoIterations,
                  double f_tolerance,
                  double tolerance, double &solution)
{
  if (f(a, R, r, Z, n, timesT)*f(b, R, r, Z, n, timesT) >= 0)
  {
    throw logic_error("** Error: Initial conditions for Bisection Method are n
    return -1;
  }
  if ((f_tolerance < 0) || (tolerance < 0) || (a > b))
  {
    throw logic_error("** Error: Fatal call of Bisection Method Routine.");
  }

  for (int i = 0; i < maxNoIterations; i++)
  {

    solution = a + (b - a) / 2;

    if ((-f_tolerance <= f(solution, R, r, Z, n, timesT))
        &&
        (f(solution, R, r, Z, n, timesT) <= f_tolerance))
    {
      maxNoIterations = i + 1;
      return 1;
    }

    // test if sol_n is close to sol_{n-1}, and if so, return
    // sol. as the solution
    if (solution != 0)
```

```

        {
            if ((solution - a) / solution < tolerance)
            {
                maxNoIterations = i + 1;
                return 2;
            }

        }

        if (f(a, R, r, Z, n, timesT)*f(solution, R, r, Z, n, timesT) > 0)
            a = solution;
        else
            b = solution;
    }

    // Bisection Method failed after maxNoIterations
    return 0;
}

int bisectionPtrF10bD(PtrF10bD f, double upto, double vect[][2], int index,
                    double R, int indexCDS_T, int indexAnteriorCDS_T,
                    double r, double Z, double *timesT, double a, double b,
                    int &maxNoIterations, double f_tolerance,
                    double tolerance, double &solution)
{
    if (f(a, upto, vect, index, R, indexCDS_T, indexAnteriorCDS_T, r, Z, timesT)
        *f(b, upto, vect, index, R, indexCDS_T, indexAnteriorCDS_T, r, Z, timesT)
        >= 0)
    {
        throw logic_error("** Error: Initial conditions for Bisection Method are n
        //exit(1);
        return -1;
    }
    if ((f_tolerance < 0) || (tolerance < 0) || (a > b))
    {
        throw logic_error("** Error: Fatal call of Bisection Method Routine. Exit!
    }

    for (int i = 0; i < maxNoIterations; i++)
    {

```

```

solution = a + (b - a) / 2;

if ((-f_tolerance <= f(solution, upto, vect, index, R, indexCDS_T,
                        indexAnteriorCDS_T, r, Z, timesT))
    &&
    (f(solution, upto, vect, index, R, indexCDS_T, indexAnteriorCDS_T, r,
        Z, timesT) <= f_tolerance))
{
    maxNoIterations = i + 1;
    return 1;
}

// test if sol_n is close to sol_{n-1}, and if so, return sol.
// as the solution
if (solution != 0)
{
    if ((solution - a) / solution < tolerance)
    {
        maxNoIterations = i + 1;
        return 2;
    }
}

if (f(a, upto, vect, index, R, indexCDS_T, indexAnteriorCDS_T, r, Z,
    timesT)
    *
    f(solution, upto, vect, index, R, indexCDS_T, indexAnteriorCDS_T, r,
        Z, timesT) > 0)
    a = solution;
else
    b = solution;
}

// Bisection Method failed after maxNoIterations
return 0;
}

double cds_pricing(double Z, double T, double R,

```

```

        double r,
        double *timesT, int noTi,
        double gamma[][2])
{

    double Ta = 0, Tc;
    int index_gamma = 1;

    double I1 = 0., I2 = 0., S;

    do
    {
        Tc = gamma[ index_gamma ][0];
        I1 += numericalIntegration_CompositeSimpson(f1, r, gamma, Ta, Tc);
        I2 += numericalIntegration_CompositeSimpson(f2, r, gamma, Ta, Tc);

        index_gamma++;
        Ta = Tc;
    }
    while (Ta < T);

    S = f_Sum(r, gamma, timesT, 1, noTi);

    return R * (I1 + S) - Z * I2;
}

double cds_quote(double Z, double T,
                 double r,
                 double *timesT, int noTi,
                 double gamma[][2])
{

    double Ta = 0, Tc;
    int index_gamma = 1;

    double I1 = 0., I2 = 0., S;

    do
    {
        Tc = gamma[ index_gamma ][0];
        //cout << "Ta: " << Ta << ", Tc: " << Tc << endl;

```

```

        //cout << f1(r, gamma, (Ta+Tc)/2) << endl;
        I1 += numericalIntegration_CompositeSimpson(f1, r, gamma, Ta, Tc);
        I2 += numericalIntegration_CompositeSimpson(f2, r, gamma, Ta, Tc);

        index_gamma++;
        Ta = Tc;
    }
    while (Ta < T);

    S = f_Sum(r, gamma, timesT, 1, noTi);

    //cout << I1 << " " << I2 << " " << S << endl;

    return (Z * I2) / (I1 + S);
}

//*****
//***** CDS CALIBRATION *****
//*****

// CALIBRARE
// calculeaza primul termen - corespunzator unei intensitati constante
// (piecewise constant) - din formula de pricing CDS
double Interval1_gamma1Ti(double b, double R, double r, double Z, int n,
                           double *timesT)
{
    double gamma1[PIECEWISE_NO][2];

    double T_0 = 0.;// aici ar trebui afectata nu valoarea 0, ci timesT[0]
    double T_n = timesT[n];

    gamma1[0][0] = T_0;
    gamma1[0][1] = b;
    gamma1[1][0] = T_n;
    gamma1[1][1] = b;

```

```

double I1 = numericalIntegration_CompositeSimpson(f1, r, gamma1, T_0, T_n);

double S = f_Sum(r, gamma1, timesT, 1, n);

double I2 = numericalIntegration_CompositeSimpson(f2, r, gamma1, T_0, T_n);

return R * I1 + R * S - Z * I2;
}

// CALIBRARE
// calculeaza al i-lea termen din formula de pricing CDS
// PARTICULARIZARE
double Interval_gamma(double y,
                      double up_to,
                      double _gamma[][2], int indexCDS,
                      double R, int indexCDS_T, int indexAnteriorCDS_T,
                      double r, double Z, double *timesT)
{

    _gamma[indexCDS][0] = timesT[indexCDS_T];
    _gamma[indexCDS][1] = y;

    double x1 = _gamma[indexCDS - 1][0];
    double x2 = _gamma[indexCDS][0];

    double I1 = numericalIntegration_CompositeSimpson(f1, r, _gamma, x1, x2);

    double S = f_Sum(r, _gamma, timesT, indexAnteriorCDS_T + 1, indexCDS_T);

    double I2 = numericalIntegration_CompositeSimpson(f2, r, _gamma, x1, x2);

    return up_to + R * I1 + R * S - Z * I2;
}

////////////////////////////////////

// particularizare la timesT !!!
int getT_index(double Ti, double *_timesT, int dim)
{
    for (int i = 1; i <= dim; i++)

```

```

        if (Ti == _timesT[i])
            return i;
    return 0;
}

int cds_plini_cali(double r, double Z,
                  int n, double *timesT,
                  int noCDS, double arrayCDS[][2],
                  double gamma[][2],
                  double f_tolerance,
                  double tolerance,
                  int maxNoIterations)
{
    int flag;
    double solution;
    int i;

    for (i = 0; i < PIECEWISE_NO; i++) gamma[i][0] = gamma[i][1] = 0;

    int indexAnteriorCDS_T = getT_index(arrayCDS[0][0], timesT, n);

    double leftBoundForBisection = 0., rightBoundForBisection = 1.;

    maxNoIterations = 50;

    flag = bisectionPtrF6D(Interval1_gamma1Ti, arrayCDS[0][1], r, Z,
                          indexAnteriorCDS_T, timesT, leftBoundForBisection,
                          rightBoundForBisection, maxNoIterations,
                          f_tolerance, tolerance, solution);

    if (flag == 0)
    {
        cout << "Bisection Method failed!" << " (" << maxNoIterations
              << " iters)" << endl;
        exit(1);
    }
}

```

```

gamma[0][0] = 0.;
gamma[0][1] = solution;
gamma[1][0] = arrayCDS[0][0];
gamma[1][1] = solution;
// y1 is computed

int i_gamma;
for (i_gamma = 2; i_gamma <= noCDS; i_gamma++)
{
    double I1 = 0., I2 = 0., S;

    int indexAnteriorCDS_T = 0;
    int indexCDS_T = getT_index(arrayCDS[i_gamma - 2][0], timesT, n);

    S = f_Sum(r, gamma, timesT, 1, indexCDS_T);

    int iTa = 0;
    int iTc;
    for (int j = 1; j < i_gamma; j++)
    {
        iTc = getT_index(arrayCDS[j - 1][0], timesT, n);
        double Ta = timesT[iTa];
        double Tc = timesT[iTc];
        //cout << "    on: [" << Ta << ", " << Tc << "], j=" << j << endl;

        I1 += numericalIntegration_CompositeSimpson(f1, r, gamma, Ta, Tc);
        I2 += numericalIntegration_CompositeSimpson(f2, r, gamma, Ta, Tc);
        //cout << "j=" << j << endl;

        iTa = iTc;
    }

    double R = arrayCDS[i_gamma - 1][1];
    double _upto = R * I1 + R * S - Z * I2;

    indexAnteriorCDS_T = indexCDS_T;
}

```



```

indexCDS_T = getT_index(arrayCDS[i_gamma - 1][0], timesT, n);

leftBoundForBisection = 0.;
rightBoundForBisection = 1.;
maxNoIterations = 50;

flag = bisectionPtrF10bD(Interval_gamma, _upto, gamma, i_gamma, R,
                        indexCDS_T, indexAnteriorCDS_T, r, Z, timesT,
                        leftBoundForBisection,
                        rightBoundForBisection, maxNoIterations,
                        f_tolerance, tolerance, solution);

if (flag == 0)
{
    cout << "Bisection Method failed!" << " (" << maxNoIterations
        << " iters)" << endl;
    exit(1);
}

}
return 0;
}

#endif //PremiaCurrentVersion

```