Help

```
extern "C" {
#include "
href../../../mod/bs1d_default/bs1d_default_stdr/bs1d_default_stdr_h_src.pdfbs1d_
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_tridiag_matrix.h"
}

struct Spline {
int n;
double * x;
double * y;
double * miu;
double * h;
double * k;
double * H;
bool linear_tail;
PnlTridiagMat* M;
PnlVect *B, *X;
void moments();
void Set_coeff();
void Set_LU();
short int down_0;
short int up_0;
short int med_0;
short int n2_0;
double* B_array;
double f6;
double *coeff_1;
double *coeff_2;
double *coeff_3;
bool LU_shared;
bool not_a_knot;
int nm1;
int nm2;
int nm3;

PnlTridiagMatLU* LU;
```

```
};


static void Spline_Distruggi(Spline S) {
free(S.coeff_1);
free(S.coeff_2);
free(S.coeff_3);
free(S.miu);
free(S.h);
free(S.k);
free(S.H);
pnl_tridiag_mat_free(&(S.M));
pnl_tridiag_mat_lu_free(&(S.LU));
pnl_vect_free(&(S.B));
pnl_vect_free(&(S.X));

}

static double Spline_Evaluate(Spline S, double x_) {

int down = 0;
int up = S.nm1;
int med = (int)(round(up*0.5));
bool out_up = 0;
bool out_dw = 0;

int n = S.n;
double* x = S.x;
double* y = S.y;
double* coeff_1 = S.coeff_1;
double* coeff_2 = S.coeff_2;
double* coeff_3 = S.coeff_3;

double* k = S.k;
double* h = S.h;

double aux, dx;

if (x_<x[0]) { out_dw = 1; down = 0; up = 1; }
else if (x_>x[n - 1]) { out_up = 1; down = n - 2; up = n - 1; }
else {
```

```
while (up - down != 1) {
if (x[med] <= x_) {
down = med;
med = (int)((up + down)*0.5);
}
else {
up = med;
med = (int)((up + down)*0.5);
}
}
}
if (S.linear_tail) {
if (out_up) { aux = y[n - 1] + (x_ - x[n - 1])*k[n - 2] / h[n - 2]; }
else if (out_dw) { aux = y[0] + (x_ - x[0])*k[0] / h[0]; }
else {
dx = x_ - x[down];
aux = y[down] + dx * (coeff_1[down] + dx * (coeff_2[down] + dx * coeff_3[down]))
}
}
else {
dx = x_ - x[down];
aux = y[down] + dx * (coeff_1[down] + dx * (coeff_2[down] + dx * coeff_3[down]))
}
return aux;
}

static void Spline_Moments(Spline S) {
double* Xa = S.X->array;
double* Ba = S.B->array;
double* miu = S.miu;
double* coeff_1 = S.coeff_1;
double* coeff_2 = S.coeff_2;
double* coeff_3 = S.coeff_3;

int nm1 = S.nm1;
int nm2 = S.nm2;
double* k = S.k;
double* h = S.h;

for (int i = 0; i<nm2; i++) {
Ba[i] = 6 * (k[i + 1] / h[i + 1] - k[i] / h[i]);
```

```
}
pnl_tridiag_mat_lu_syslin(S.X, S.LU, S.B);

for (int i = 1; i<nm1; i++) {
miu[i] = Xa[i - 1];
}
if (S.not_a_knot) {
miu[0] = miu[1] - (h[0] / h[1])*(miu[2] - miu[1]);
miu[nm1] = miu[nm2] + (h[nm2] / h[nm2 - 1])*(miu[nm2] - miu[nm2 - 1]);
}
else {
miu[0] = 0; miu[nm1] = 0;
}

for (int i = 0; i<nm1; i++) {
coeff_1[i] = (k[i] / h[i] - (miu[i + 1] + 2 * miu[i])*h[i] / 6.0);
coeff_2[i] = 0.5*miu[i];
coeff_3[i] = (miu[i + 1] - miu[i]) / (6.0*h[i]);
}

}

void Spline_Set_LU(Spline S) {
double* DL = S.M->DL;
double* D = S.M->D;
double* DU = S.M->DU;
double* h = S.h;
double* H = S.H;
int nm3 = S.nm3;

for (int i = 0; i<nm3; i++) {
DL[i] = h[i + 1];
D[i] = H[i];
DU[i] = h[i + 1];
}

D[nm3] = H[nm3];
if (S.not_a_knot) {
D[0] += h[0] + SQR(h[0]) / h[1];
DU[0] += (-SQR(h[0]) / h[1]);
DL[nm3 - 1] += (-SQR(h[nm3]) / h[nm3 - 1]);
```

```
D[nm3] += h[nm3] + SQR(h[nm3]) / h[nm3 - 1];
}
pnl_tridiag_mat_lu_compute(S.LU, S.M);
}

static Spline Create_Spline(int n, double* x, double* y, bool l) {
Spline S;
int nm1 = n - 1;
int nm2 = n - 2;
S.n = n;
S.x = x;
S.y = y;
S.miu = NULL;
S.linear_tail = l;
S.LU_shared = 0;
S.nm1 = n - 1;
S.nm2 = n - 2;
S.nm3 = n - 3;
S.h = (double *)malloc(nm1 * sizeof(double));
S.k = (double *)malloc(nm1 * sizeof(double));
S.not_a_knot = 0;
for (int i = 0; i<n - 1; i++) { S.h[i] = x[i + 1] - x[i]; S.k[i] = y[i + 1] - y[
S.M = pnl_tridiag_mat_create(n - 2);
S.LU = pnl_tridiag_mat_lu_create(n - 2);
S.B = pnl_vect_create(n - 2); S.B_array = S.B->array;
S.X = pnl_vect_create(n - 2);
S.miu = (double *)malloc(n * sizeof(double));
S.H   = (double *)malloc(nm2 * sizeof(double));
S.coeff_1 = (double *)malloc(nm1 * sizeof(double));
S.coeff_2 = (double *)malloc(nm1 * sizeof(double));
S.coeff_3 = (double *)malloc(nm1 * sizeof(double));
for (int i = 0; i<n - 2; i++) {
S.H[i] = 2 * (S.h[i] + S.h[i + 1]);
}
Spline_Set_LU(S);
Spline_Moments(S);
S.down_0 = 0;
S.up_0 = n - 1;
S.med_0 = (int)(S.up_0*0.5);
S.n2_0 = n - 2; S.f6 = 1 / 6.0;
```

```
return S;
}

struct BS_EDP {

int N;
int J;
double *L_input;
double *Y_input;
double *P_input;
bool Neumann;
double dt;
double dL;
double T_max;
double a_CN;
double cdL;
double cdL2;
double cd0;
bool correct_small_s;
double small_s_limit;
bool AM;
PnlTridiagMat *Matrix;
PnlTridiagMatLU *LU;
PnlVect *B;
PnlVect *X;
double aux_c0, aux_c1, aux_c2, aux_c3, aux_c4, aux_c5;
double aux_d0, aux_d1, aux_d2, aux_d3;
double aux_u0, aux_u1, aux_u2, aux_u3;
};


static void EDP_Set_LU(BS_EDP &EDP) {

int J = EDP.J;
bool Neumann = EDP.Neumann;
double dt = EDP.dt;
double dL = EDP.dL;
double a_CN = EDP.a_CN;
double cdL = EDP.cdL;
double cdL2 = EDP.cdL2;
double cd0 = EDP.cd0;
```

```
bool correct_small_s = EDP.correct_small_s;
double small_s_limit = EDP.small_s_limit;
PnlTridiagMat *Matrix = EDP.Matrix;
PnlTridiagMatLU *LU = EDP.LU;
double aux_c0,aux_c1, aux_c2, aux_c3, aux_c4, aux_c5;
double aux_d0, aux_d1, aux_d2, aux_d3;
double aux_u0, aux_u1, aux_u2, aux_u3;

double* DL = Matrix->DL;
double* D = Matrix->D;
double* DU = Matrix->DU;

bool print = 0;
bool small_s = (small_s_limit*ABS(cdL + cdL2)>cdL2);
bool up_str = (cdL>0);
double aux = cdL;
double aux2 = dt / dL;
double aux3 = aux2 / dL;
double aux4 = 0.5*dt / dL;


aux_c0 = (1 - a_CN)*(aux*aux4 + cdL2 * aux3);
aux_c1 = -1 - (1 - a_CN)*(2.0*cdL2*aux3 + cd0 * dt);
aux_c2 = (1 - a_CN)*(-aux * aux4 + cdL2 * aux3);
aux_c3 = -a_CN * (aux*aux4 + cdL2 * aux3);
aux_c4 = -1 + a_CN * (2.0*cdL2*aux3 + cd0 * dt);
aux_c5 = -a_CN * (-aux * aux4 + cdL2 * aux3);
if (correct_small_s&&small_s) {
if (print) { printf("correct small s: "); }
if (up_str) {
if (print) { printf(" up_stream\ n"); }
aux_c0 = (1 - a_CN)*(aux*aux2 + cdL2 * aux3);
aux_c1 = -1 - (1 - a_CN)*(2.0*cdL2*aux3 + cd0 * dt + aux * aux2);
aux_c2 = (1 - a_CN)*(cdL2*aux3);
aux_c3 = -a_CN * (aux*aux2 + cdL2 * aux3);
aux_c4 = -1 + a_CN * (2.0*cdL2*aux3 + cd0 * dt + aux * aux2);
aux_c5 = -a_CN * (cdL2*aux3);
}
else {
if (print) { printf(" down_stream\ n"); }
aux_c0 = (1 - a_CN)*(cdL2*aux3);
```

```
aux_c1 = -1 - (1 - a_CN)*(2.0*cdL2*aux3 + cd0 * dt - aux * aux2);
aux_c2 = (1 - a_CN)*(-aux * aux2 + cdL2 * aux3);
aux_c3 = -a_CN * (cdL2*aux3);
aux_c4 = -1 + a_CN * (2.0*cdL2*aux3 + cd0 * dt - aux * aux2);
aux_c5 = -a_CN * (-aux * dt / (dL)+cdL2 * aux3);
}
}

if (Neumann) {
aux_d0 = (1 - a_CN);
aux_d1 = -(1 - a_CN);
aux_d2 = -a_CN;
aux_d3 = a_CN;
}
else {
aux_d0 = (1 - a_CN)*aux*(dt / dL);
aux_d1 = -1 - (1 - a_CN)*(aux*dt / dL + cd0 * dt);
aux_d2 = -a_CN * aux*dt / dL;
aux_d3 = -1 + a_CN * (dt*aux / dL + cd0 * dt);
}

if (Neumann) {
aux_u0 = (1 - a_CN);
aux_u1 = -(1 - a_CN);
aux_u2 = -a_CN;
aux_u3 = a_CN;
}
else {
aux_u0 = -1 + (1 - a_CN)*(dt*aux / dL - dt * cd0);
aux_u1 = -(1 - a_CN)*dt*aux / dL;
aux_u2 = -1 - a_CN * (aux*dt / dL - cd0 * dt);
aux_u3 = a_CN * dt*aux / dL;
}

for (int j = 0; j<J - 1; j++) {
DL[j] = aux_c2;
D[j] = aux_c1;
DU[j] = aux_c0;
}
D[0] = aux_d1;
DU[0] = aux_d0;
```

```
D[J - 1] = aux_u0;
DL[J - 2] = aux_u1;

pnl_tridiag_mat_lu_compute(LU, Matrix);

EDP.aux_c0 = aux_c0;
EDP.aux_c1 = aux_c1;
EDP.aux_c2 = aux_c2;
EDP.aux_c3 = aux_c3;
EDP.aux_c4 = aux_c4;
EDP.aux_c5 = aux_c5;

EDP.aux_d0 = aux_d0;
EDP.aux_d1 = aux_d1;
EDP.aux_d2 = aux_d2;
EDP.aux_d3 = aux_d3;

EDP.aux_u0 = aux_u0;
EDP.aux_u1 = aux_u1;
EDP.aux_u2 = aux_u2;
EDP.aux_u3 = aux_u3;

}

static void EDP_Compute_output(BS_EDP EDP, double * Result) {
int N = EDP.N;
int J = EDP.J;
double *Y_input = EDP.Y_input;
double *P_input = EDP.P_input;
bool AM = EDP.AM;
PnlTridiagMatLU *LU = EDP.LU;
PnlVect *B = EDP.B;
PnlVect *X = EDP.X;


double* X_array = X->array;
double* B_array = B->array;

double aux_c_3 = EDP.aux_c3;
double aux_c_4 = EDP.aux_c4;
double aux_c_5 = EDP.aux_c5;
```

```
double aux_d_2 = EDP.aux_d2;
double aux_d_3 = EDP.aux_d3;

double aux_u_2 = EDP.aux_u2;
double aux_u_3 = EDP.aux_u3;

int Jm1 = J - 1;
int Jm2 = J - 2;

for (int j = 0; j<J; j++) {
X_array[j] = Y_input[j];
}

for (int n = N; n>0; n--) {
for (int j = 1; j<J - 1; j++) {
B_array[j] = aux_c_3 * X_array[j + 1] + aux_c_4 * X_array[j] + aux_c_5 * X_array
}

B_array[0] = aux_d_2 * X_array[1] + aux_d_3 * X_array[0];
B_array[J - 1] = aux_u_2 * X_array[Jm1] + aux_u_3 * X_array[Jm2];
pnl_tridiag_mat_lu_syslin(X, LU, B);

if (AM) {
for (int j = 0; j<J; j++) { X_array[j] = MAX(X_array[j], P_input[j]); }
}
}

for (int j = 0; j<J; j++) {
Result[j] = X_array[j];
}
}

void EDP_Distruggi(BS_EDP &EDP) {

PnlTridiagMatLU *LU = EDP.LU;
PnlVect *B = EDP.B;
PnlVect *X = EDP.X;

pnl_tridiag_mat_free(&(EDP.Matrix));
pnl_tridiag_mat_lu_free(&LU);
```

```
pnl_vect_free(&X);
pnl_vect_free(&B);

}

BS_EDP Create_BS_EDP(const int N_, const int J_, double* L_input_, double* Y_inp
BS_EDP EDP;
double aux;
EDP.T_max = T_max_;
EDP.N = N_;
EDP.J = J_;
EDP.cdL = cdL_;
EDP.cdL2 = cdL2_;
EDP.cd0 = cd0_;
EDP.a_CN = a_CN_;
EDP.AM = AM_;
EDP.correct_small_s = corr_;
EDP.small_s_limit = limit;
EDP.Y_input = Y_input_;
EDP.L_input = L_input_;
EDP.P_input = P_input_;
EDP.dt = EDP.T_max / (0.0 + EDP.N);
EDP.dL = EDP.L_input[1] - EDP.L_input[0];
aux = SQR(EDP.dL) / (2.0*EDP.cdL2);
EDP.Matrix = pnl_tridiag_mat_create(EDP.J);
EDP.LU = pnl_tridiag_mat_lu_create(EDP.J);
EDP.X = pnl_vect_create(EDP.J);
EDP.B = pnl_vect_create(EDP.J);
EDP.Neumann = Neumann_;
EDP_Set_LU(EDP);
return EDP;
}


extern "C" {
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2019+2) //The "#els

static int CHK_OPT(FD_CVAMixedPDE)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
```

```
int CALC(FD_CVAMixedPDE)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

static int CVA_Black_Scholes(int am,double Default_intensity,double Recovery_Rat
{
   double r = Black_Scholes_interest_rate_r;
double d = Black_Scholes_dividend_rate_d;
double s = Black_Scholes_volatility_sigma;
double S_0 = Black_Scholes_initial_price_S0;
double T = Option_maturity;
bool AM = am;
double RR = Recovery_Rate;
double DI = Default_intensity;
int N = time_steps;
int J = space_steps;


double alpha = 0.5;
double Factor = 1000;
double dt = T / (N + 0.0);
double DIdt = DI * dt;
double factor, CVA, Pr1, Pr2;
double *L,*S,*Y,*Y2,*Po;
double L_min = log(S_0) - log(Factor);
double L_max = log(S_0) + log(Factor);
double dL = (L_max - L_min) / (0.0 + J - 1);
L =  (double *)malloc(J * sizeof(double));
S =  (double *)malloc(J * sizeof(double));
Y =  (double *)malloc(J * sizeof(double));
Y2 = (double *)malloc(J * sizeof(double));
Po = (double *)malloc(J * sizeof(double));


factor = RR * (exp(-DI * (T - dt)) - exp(-DI * T));
factor = factor + (exp(-DI * T));

for (int i = 0; i<J; i++) {
L[i] = L_min + i * dL;
```

```
S[i] = exp(L[i]);
Y[i] = (p->Compute)(p->Par, S[i]);
Y2[i] = Y[i] * factor;
Po[i] = Y[i];
}

BS_EDP EDP_Risk_free = Create_BS_EDP(1, J, L, Y , alpha, dt, r - d - SQR(s)*0.5,
BS_EDP EDP_Risky = Create_BS_EDP(1, J, L, Y2, alpha, dt, r - d - SQR(s)*0.5, SQR

for (int n = N - 1; n > -1; n--) {
factor = RR * (exp(-DIdt *(n - 1)) - exp(-DIdt*(n)));
EDP_Compute_output(EDP_Risk_free, Y);
for (int i = 0; i<J; i++) {
Y2[i] += Y[i] * factor;
}
EDP_Compute_output(EDP_Risky, Y2);
}

Spline Sp = Create_Spline(J, S, Y, 0);
Spline Sp2 = Create_Spline(J, S, Y2, 0);
Pr1 = Spline_Evaluate(Sp,S_0);
Pr2 = Spline_Evaluate(Sp2, S_0);
CVA = Pr1 - Pr2;

free(L);
free(Y);
free(Y2);
free(Po);

Spline_Distruggi(Sp);
Spline_Distruggi(Sp2);
EDP_Distruggi(EDP_Risk_free);
EDP_Distruggi(EDP_Risky);

*ptprice=CVA;

return OK;
}

static int CALC(FD_CVAMixedPDE)(void *Opt, void *Mod, PricingMethod *Met)
{
```

```
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r, divid;

  r = ptMod->Interest.Val.V_DOUBLE;
  divid=0;

  return CVA_Black_Scholes(ptOpt->EuOrAm.Val.V_BOOL,ptMod->Intensity.Val.V_PDOUB

}

static int CHK_OPT(FD_CVAMixedPDE)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "CVA_CallEuro") == 0)||(strcmp(((Option *)O
    return OK;
  return  WRONG;
}
#endif//PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->HelpFilenameHint = "FD_CVAMixedPDE";
      Met->Par[0].Val.V_PINT = 100;
  Met->Par[1].Val.V_PINT = 100;
    }
  return OK;
}


PricingMethod MET(FD_CVAMixedPDE) =
{
  "FD_CVAMixedPDE",
  {
    {"Numbers of Time Steps", PINT, {22}, ALLOW},
{"Numbers of Space Steps", PINT, {22}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_CVAMixedPDE),
```

```
  {
    {"CVA", DOUBLE, {0}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(FD_CVAMixedPDE),
  CHK_ok,
  MET(Init)
};
}
```