Help

```
extern "C" {
#include  "
href../../../mod/merhes1d_default/merhes1d_default_stdr/merhes1d_default_stdr_h_
}

#include "
href../../../common/math/fft_h_src.pdfmath/fft.h"
#include "
href../../../common/math/numerics_h_src.pdfmath/numerics.h"
#include "
href../../../common/math/levy_h_src.pdfmath/levy.h"
extern "C" {
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2019+2) //The "#els
static int CHK_OPT(FD_CVAMixedPDEBates)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(FD_CVAMixedPDEBates)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

////////////////////////////////////////////////////////////////////////////////
//Compute CVA.
static int NSpace;
static double **V, **P_old, **P_new, ***P, ***P2, ***vect_s, ***vect_s2;
static double **f;
static int **f_down, **f_up;
static double **pu_f, **pd_f;
static double *vect_y;


static int memory_allocation(int Nt, int N)
{
int i, j;

vect_y = (double *)malloc((N + 1) * sizeof(double));
```

```
V = (double **)calloc(Nt + 1, sizeof(double *));
if (V == NULL)
return MEMORY_ALLOCATION_FAILURE;
for (i = 0; i < Nt + 1; i++)
{
V[i] = (double *)calloc(Nt + 1, sizeof(double));
if (V[i] == NULL)
return MEMORY_ALLOCATION_FAILURE;
}

pu_f = (double **)calloc(Nt + 1, sizeof(double *));
if (pu_f == NULL)
return MEMORY_ALLOCATION_FAILURE;
for (i = 0; i < Nt + 1; i++)
{
pu_f[i] = (double *)calloc(Nt + 1, sizeof(double));
if (pu_f[i] == NULL)
return MEMORY_ALLOCATION_FAILURE;
}

pd_f = (double **)calloc(Nt + 1, sizeof(double *));
if (pd_f == NULL)
return MEMORY_ALLOCATION_FAILURE;
for (i = 0; i < Nt + 1; i++)
{
pd_f[i] = (double *)calloc(Nt + 1, sizeof(double));
if (pd_f[i] == NULL)
return MEMORY_ALLOCATION_FAILURE;
}

f = (double **)calloc(Nt + 1, sizeof(double *));
if (f == NULL)
return MEMORY_ALLOCATION_FAILURE;
for (i = 0; i < Nt + 1; i++)
{
f[i] = (double *)calloc(Nt + 1, sizeof(double));
if (f[i] == NULL)
return MEMORY_ALLOCATION_FAILURE;
}

f_down = (int **)calloc(Nt + 1, sizeof(int *));
```

```
if (f_down == NULL)
return MEMORY_ALLOCATION_FAILURE;
for (i = 0; i < Nt + 1; i++)
{
f_down[i] = (int *)calloc(Nt + 1, sizeof(int));
if (f_down[i] == NULL)
return MEMORY_ALLOCATION_FAILURE;
}

f_up = (int **)calloc(Nt + 1, sizeof(int *));
if (f_up == NULL)
return MEMORY_ALLOCATION_FAILURE;
for (i = 0; i < Nt + 1; i++)
{
f_up[i] = (int *)calloc(Nt + 1, sizeof(int));
if (f_up[i] == NULL)
return MEMORY_ALLOCATION_FAILURE;
}

P_old = (double **)malloc((N + 1) * sizeof(double *));
for (i = 0; i <= N; i++)
P_old[i] = (double *)malloc((Nt + 1) * sizeof(double));

P_new = (double **)malloc((N + 1) * sizeof(double *));
for (i = 0; i <= N; i++)
P_new[i] = (double *)malloc((Nt + 1) * sizeof(double));

P = (double ***)malloc((Nt + 1) * sizeof(double**));
for (i = 0; i <= Nt; i++)
P[i] = (double **)malloc((N + 1) * sizeof(double*));
for (i = 0; i <= Nt; i++)
for (j = 0; j <= N; j++)
P[i][j] = (double *)malloc((Nt + 1) * sizeof(double));

P2 = (double ***)malloc((Nt + 1) * sizeof(double**));
for (i = 0; i <= Nt; i++)
P2[i] = (double **)malloc((N + 1) * sizeof(double*));
for (i = 0; i <= Nt; i++)
for (j = 0; j <= N; j++)
P2[i][j] = (double *)malloc((Nt + 1) * sizeof(double));
```

```
vect_s = (double ***)malloc((Nt + 1) * sizeof(double**));
for (i = 0; i <= Nt; i++)
vect_s[i] = (double **)malloc((N + 1) * sizeof(double*));
for (i = 0; i <= Nt; i++)
for (j = 0; j <= N; j++)
vect_s[i][j] = (double *)malloc((Nt + 1) * sizeof(double));

vect_s2 = (double ***)malloc((Nt + 1) * sizeof(double**));
for (i = 0; i <= Nt; i++)
vect_s2[i] = (double **)malloc((N + 1) * sizeof(double*));
for (i = 0; i <= Nt; i++)
for (j = 0; j <= N; j++)
vect_s2[i][j] = (double *)malloc((Nt + 1) * sizeof(double));

return OK;
}

static void free_memory(int Nt, int N)
{
int i, j;

free(vect_y);

for (i = 0; i < Nt + 1; i++)
free(V[i]);
free(V);

for (i = 0; i < Nt + 1; i++)
free(pu_f[i]);
free(pu_f);

for (i = 0; i < Nt + 1; i++)
free(pd_f[i]);
free(pd_f);

for (i = 0; i < Nt + 1; i++)
free(f[i]);
free(f);

for (i = 0; i < Nt + 1; i++)
free(f_up[i]);
```

```
free(f_up);

for (i = 0; i < Nt + 1; i++)
free(f_down[i]);
free(f_down);

for (i = 0; i < N + 1; i++)
free(P_old[i]);
free(P_old);

for (i = 0; i < N + 1; i++)
free(P_new[i]);
free(P_new);

for (i = 0; i<Nt + 1; i++)
for (j = 0; j<N + 1; j++)
free(P[i][j]);
for (j = 0; j<Nt + 1; j++)
free(P[j]);
free(P);

for (i = 0; i<Nt + 1; i++)
for (j = 0; j<N + 1; j++)
free(P2[i][j]);
for (j = 0; j<Nt + 1; j++)
free(P2[j]);
free(P2);

for (i = 0; i<Nt + 1; i++)
for (j = 0; j<N + 1; j++)
free(vect_s[i][j]);
for (j = 0; j<Nt + 1; j++)
free(vect_s[j]);
free(vect_s);

for (i = 0; i<Nt + 1; i++)
for (j = 0; j<N + 1; j++)
free(vect_s2[i][j]);
for (j = 0; j<Nt + 1; j++)
free(vect_s2[j]);
free(vect_s2);
```

```
return;
}

static double compute_f(double r, double omega)
{
return 2.*sqrt(r) / omega;
}

static double compute_v(double R, double omega)
{
double val;

val = SQR(R) * SQR(omega) / 4.;
if (R > 0.)
val = SQR(R) * SQR(omega) / 4.;
else
val = 0.0;
return val;
}

static double compute_S(double Y, double rv, double omega, double rho)
{
double val;

val = exp(Y) * exp(rho * rv / omega);

return val;
}

static int tree_v(double tt, double v0, double kappa, double theta, double omega
{
int i, j;
int z;
double Ru, Rd;
double mu_r, v_curr;
double dt, sqrt_dt;

/*Fixed tree for R=f*/
f[0][0] = compute_f(v0, omega);
```

```
dt = tt / (double)Nt;
sqrt_dt = sqrt(dt);

V[0][0] = compute_v(f[0][0], omega);
f[1][0] = f[0][0] - sqrt_dt;
f[1][1] = f[0][0] + sqrt_dt;
V[1][0] = compute_v(f[1][0], omega);
V[1][1] = compute_v(f[1][1], omega);
for (i = 1; i < Nt; i++)
for (j = 0; j <= i; j++)
{
f[i + 1][j] = f[i][j] - sqrt_dt;
f[i + 1][j + 1] = f[i][j] + sqrt_dt;
V[i + 1][j] = compute_v(f[i + 1][j], omega);
V[i + 1][j + 1] = compute_v(f[i + 1][j + 1], omega);
}

/*Evolve tree for f*/
for (i = 0; i < Nt; i++)
{
for (j = 0; j <= i; j++)
{
/*Compute mu_f*/
v_curr = V[i][j];

mu_r = kappa * (theta - v_curr);

z = 0;
while ((V[i][j] + mu_r * dt < V[i + 1][j - z])
&& (j - z >= 0))
{

z = z + 1;
}
f_down[i][j] = -z;
Rd = V[i + 1][j - z];


if (z > 0)
z = 0;
```

```
else z = 1;

while ((V[i][j] + mu_r * dt > V[i + 1][j + z])
&& (j + z <= i))
{
z = z + 1;
}

Ru = V[i + 1][j + z];

f_up[i][j] = z;
pu_f[i][j] = (V[i][j] + mu_r * dt - Rd) / (Ru - Rd);

if ((Ru - 1.e-9 > V[i + 1][i + 1]) || (j + f_up[i][j] > i + 1))
{
pu_f[i][j] = 1;

f_up[i][j] = i + 1 - j;
f_down[i][j] = i - j;
}

if ((Rd + 1.e-9 < V[i + 1][0]) || (j + f_down[i][j] < 0))
{
pu_f[i][j] = 0.;
f_up[i][j] = 1 - j;
f_down[i][j] = 0 - j;
}
pd_f[i][j] = 1. - pu_f[i][j];

}
}

return 1;
}

static  int FDHYBRIDTREE_Bates(int call_or_put, int am, double K, double T, doub
{ // First PIDE

//Sigma for boundary condition
double sigma = 0.5;
```

```
/*Construction of the model*/
double delta = sqrt(gamma2);
Merton_measure measure(mean, delta, lambda, sigma, dx);


double Al = log(2. / 3) + T * measure.espX1 - km * sqrt(T * measure.varX1);
double Ar = log(2.) + r * T + km * sqrt(T * measure.varX1);
if (Al < -30) Al = -30;
if (Ar > 30) Ar = 30;
int Nl = (int)ceil(-Al / dx);
int Nr = (int)ceil(Ar / dx);
NSpace = Nl + Nr;
Al = -Nl * dx;
Ar = Nr * dx;
Al = log(S0) - rho / omega * v0 + Al;
Ar = log(S0) - rho / omega * v0 + Ar;

int k, i, j;
int fv_up, fv_down;
double stock;
double dt;
double **P_Old, **P_New;
double *A, *B, *C, *S;
double *beta_p, *u, *v, *Pr;
int N_fft;
int p;
int NN_fft;
int Nz;
double *mu;
double *mu_img;
double *uaux;
double *uaux_img;
double *somme;
double *somme_img;
double *vect_t;
double *vect_y;
double y0;
double *Price;

/* Memory Allocation */
memory_allocation(Ntime, NSpace);
```

```
/* Tree construction for v */

tree_v(T, v0, kappa, theta, omega, Ntime);
dt = T / (double)Ntime;

dx = (Ar - Al) / (NSpace);

const int Kmax = measure.Kmax;
const int Kmin = measure.Kmin;

int Ntp1 = Ntime + 1;
int NSp1 = NSpace + 1;

vect_t = (double*)malloc(Ntp1*sizeof(double));
vect_y = (double*)malloc(NSp1 * sizeof(double));

for (int n = 0; n <= Ntime; n++)
vect_t[n] = n * dt;

P_Old = (double**)malloc(Ntp1 * sizeof(double*));
for (int i = 0; i <= Ntime; i++)
P_Old[i] = (double*)malloc(NSp1 * sizeof(double));

P_New = (double**)malloc(Ntp1 * sizeof(double*));
for (int i = 0; i <= Ntime; i++)
P_New[i] = (double*)malloc(NSp1 * sizeof(double));

//Terminal Condition
for (j = 0; j <= NSpace; j++)
vect_y[j] = Al + (double)j * dx;

//Mesh of S
for (i = Ntime; i >= 0; i--)
for (k = 0; k <= i; k++)
for (j = 0; j <= NSpace; j++)
{
vect_s[i][j][k] = compute_S(vect_y[j], V[i][k], omega, rho);
}

for (k = 0; k <= Ntime; k++)
```

```
for (int i = 0; i <= NSpace; i++)
{
stock = compute_S(vect_y[i], V[Ntime][k], omega, rho);
if (call_or_put)
P_Old[k][i] = MAX(0, stock - K);
else P_Old[k][i] = MAX(0, K - stock);
P[Ntime][i][k] = P_Old[k][i];

//printf("%f %f %f\ n",vect_y[i],V[Ntime][k],stock);
}

//some useful coefficients
/*matrix coefficients of the implicit part*/

A = (double*)malloc(NSp1 * sizeof(double));
B = (double*)malloc(NSp1 * sizeof(double));
C = (double*)malloc(NSp1 * sizeof(double));
S = (double*)malloc(NSp1 * sizeof(double));
Price = (double*)malloc(NSp1 * sizeof(double));
Pr = (double*)malloc(NSp1 * sizeof(double));
beta_p = (double*)malloc(NSp1 * sizeof(double));
u = (double*)malloc(NSp1 * sizeof(double));
v = (double*)malloc(NSp1 * sizeof(double));


N_fft = NSpace + Kmax - Kmin; //number of non-zero values of u involved in compu
  //zero-padding to obtain NN = N + Nz = 2^p
p = 1;
NN_fft = 2; //size of auxiliary vectors
while (NN_fft < N_fft)
{
p++;
NN_fft = 2 * NN_fft;
}
Nz = NN_fft - N_fft; // number of extra zeros

mu = (double*)malloc(NN_fft * sizeof(double));
mu_img = (double*)malloc(NN_fft * sizeof(double));
uaux = (double*)malloc(NN_fft * sizeof(double));
uaux_img = (double*)malloc(NN_fft * sizeof(double));
somme = (double*)malloc(NN_fft * sizeof(double));
```

```
somme_img = (double*)malloc(NN_fft * sizeof(double));

/*computation of the Fourier transform of nu*/
for (int i = 0; i < Kmax - Kmin + 1; i++)
{
mu[i] = (*measure.nu_array)[Kmax - Kmin - i];
mu_img[i] = 0;
}
for (int i = Kmax - Kmin + 1; i < NN_fft; i++)
{
mu[i] = 0;
mu_img[i] = 0;
}
fft1d(mu, mu_img, NN_fft, -1);

double discount = exp(-r * dt);

for (int n = Ntime; n >= 1; n--) //time iterations
{
for (k = 0; k <= n - 1; k++)
{
fv_up = f_up[n - 1][k];
fv_down = f_down[n - 1][k];

double z, vv;
z = r - divid - 0.5 * V[n - 1][k] - rho * kappa * (theta - V[n - 1][k]) / omega;
vv = 0.5 * V[n - 1][k] * (1. - SQR(rho));

double alpha, beta, gamma;
int PriceIndex;
double bound1, bound2;

if (call_or_put == 1)
{
bound1 = 0;
bound2 = compute_S(vect_y[NSpace], V[Ntime][k], omega, rho) * exp(-divid * (Ntim
}
else
{
bound1 = K * exp(-r * (Ntime - k) * dt) - compute_S(vect_y[0], V[Ntime][k], omeg
bound2 = 0;
```

```
}

alpha = (-vv * dt / SQR(dx) + z * dt / (2.*dx));
beta = 1 + vv * 2 * dt / SQR(dx);
gamma = (-vv * dt / SQR(dx) - z * dt / (2 * dx));

for (PriceIndex = 1; PriceIndex <= NSpace - 1; PriceIndex++)
{
A[PriceIndex] = alpha;
B[PriceIndex] = beta;
C[PriceIndex] = gamma;
}

/*Set Gauss*/
for (PriceIndex = NSpace - 2; PriceIndex >= 1; PriceIndex--)
B[PriceIndex] = B[PriceIndex] - C[PriceIndex] * A[PriceIndex + 1] / B[PriceIndex
for (PriceIndex = 1; PriceIndex < NSpace; PriceIndex++)
A[PriceIndex] = A[PriceIndex] / B[PriceIndex];
for (PriceIndex = 1; PriceIndex < NSpace - 1; PriceIndex++)
C[PriceIndex] = C[PriceIndex] / B[PriceIndex + 1];

//Mixture of terminal conditions
for (int i = 0; i <= NSpace; i++)
Price[i] = pu_f[n - 1][k] * P_Old[k + fv_up][i] + pd_f[n - 1][k] * P_Old[k + fv_

// /*calculation of the discretized integral using FFT*/
for (int i = 0; i < NSpace - 1; i++)
{
if ((Kmax + 1 + i < 0) || (Kmax + 1 + i >= NSpace))
{
uaux[i] = 0.;
uaux[i] = MAX(0., compute_S(Al + (double)(Kmax + 1 + i) * dx, V[n - 1][k], omega
}
else uaux[i] = Price[Kmax + 1 + i];
uaux_img[i] = 0;
}

for (int i = NSpace - 1; i < NSpace + Nz - 1; i++)
{
uaux[i] = 0; //zero-padding
uaux_img[i] = 0;
```

```
}
for (int i = NSpace + Nz - 1; i < NN_fft; i++)
{
if ((Kmin - NSpace - Nz + 1 + i < 0) || (Kmin - NSpace - Nz + 1 + i >= NSpace))
{
uaux[i] = 0.;
uaux[i] = MAX(0., compute_S(Al + (double)(Kmin - NSpace - Nz + 1 + i) * dx, V[n
}
else uaux[i] = Price[Kmin - NSpace - Nz + 1 + i];
uaux_img[i] = 0;
}

fft1d(uaux, uaux_img, NN_fft, -1);

for (int i = 0; i < NN_fft; i++)
{
somme[i] = mu[i] * uaux[i] - mu_img[i] * uaux_img[i];
somme_img[i] = mu_img[i] * uaux[i] + mu[i] * uaux_img[i];
}
fft1d(somme, somme_img, NN_fft, 1);

if (measure.alpha < 0)
{
S[0] = bound1 + dt * (somme[NN_fft - 1] - measure.alpha*(Price[1] - bound1) / dx
;
for (PriceIndex = 1; PriceIndex<NSpace; PriceIndex++)
S[PriceIndex] = Price[PriceIndex] + dt * (somme[PriceIndex - 1] - measure.alpha*

S[NSpace] = bound2 + dt * (somme[NN_fft - 1] - measure.alpha*(bound2 - Price[NSp
}
else
{
S[0] = bound1 + dt * (somme[NN_fft - 1] - measure.alpha*(Price[1] - bound1) / dx
;
for (PriceIndex = 1; PriceIndex<NSpace; PriceIndex++)
S[PriceIndex] = Price[PriceIndex] + dt * (somme[PriceIndex - 1] - measure.alpha*
S[NSpace] = bound2 + dt * (somme[NN_fft - 1] - measure.alpha*(bound2 - Price[NSp
}

/*Solve the system*/
for (PriceIndex = NSpace - 1; PriceIndex >= 1; PriceIndex--)
```

```
S[PriceIndex] = S[PriceIndex] - C[PriceIndex] * S[PriceIndex + 1];

Pr[1] = S[1] / B[1];


for (PriceIndex = 2; PriceIndex < NSpace; PriceIndex++)
Pr[PriceIndex] = S[PriceIndex] / B[PriceIndex] - A[PriceIndex] * Pr[PriceIndex -

P_New[k][0] = 0;
P_New[k][NSpace] = 0;


if (am) {
if (call_or_put) {
for (PriceIndex = 1; PriceIndex < NSpace; PriceIndex++) {
stock = compute_S(vect_y[PriceIndex], V[n - 1][k], omega, rho);
P_New[k][PriceIndex] = MAX(discount * Pr[PriceIndex], MAX(0, stock - K));
}
}
else {
for (PriceIndex = 1; PriceIndex < NSpace; PriceIndex++) {
stock = compute_S(vect_y[PriceIndex], V[n - 1][k], omega, rho);
P_New[k][PriceIndex] = MAX(discount * Pr[PriceIndex], MAX(0, K - stock));
}

}
}
else {
for (PriceIndex = 1; PriceIndex < NSpace; PriceIndex++)
{
P_New[k][PriceIndex] = discount * Pr[PriceIndex];
}
}



}//k iteration

 //Copy
for (j = 0; j <= NSpace; j++) //for (j = 1; j < NSpace; j++)
for (k = 0; k <= n - 1; k++)
```

```
{
P_Old[k][j] = P_New[k][j];
P[n - 1][j][k] = P_Old[k][j];
//if(n==Ntime)
//printf("%f \ n",P_Old[k][j]);
}

}//end of time iterations


 //s0 Index
int Index;
double sm, s, sp;

y0 = log(S0) - rho / omega * V[0][0];
Index = 0.;
while (vect_y[Index] < y0)
{
Index++;
}

Index--;
if (Index < 0) Index = 0;

sm = compute_S(vect_y[Index], V[0][0], omega, rho);
s = compute_S(y0, V[0][0], omega, rho);
sp = compute_S(vect_y[Index + 1], V[0][0], omega, rho);

//Price
*ptprice = P_New[0][Index] + (P_New[0][Index + 1] - P_New[0][Index]) * (s - sm)

//Delta
//*ptdelta = (P_New[0][Index + 1] - P_New[0][Index]) / (s - sm);


free(beta_p);
free(u);
free(v);

free(mu);
free(mu_img);
```

```
free(uaux);
free(uaux_img);
free(somme);
free(somme_img);

free(A);
free(B);
free(C);
free(S);
free(Pr);

free(vect_t);
free(vect_y);
free(Price);

for (i = 0; i <= Ntime; i++)
free(P_Old[i]);

free(P_Old);

for (i = 0; i <= Ntime; i++)
free(P_New[i]);

free(P_New);


return OK;

}


  static int FDHYBRIDTREE_Bates2(int call_or_put, double K, double T, double S0,
{// Second PIDE

//Sigma for boundary condition
double sigma = 0.5;


/*Construction of the model*/
double delta = sqrt(gamma2);
Merton_measure measure(mean, delta, lambda, sigma, dx);
```

```
double Al = log(2. / 3) + T * measure.espX1 - km * sqrt(T * measure.varX1);
double Ar = log(2.) + r * T + km * sqrt(T * measure.varX1);
if (Al < -30) Al = -30;
if (Ar > 30) Ar = 30;
int Nl = (int)ceil(-Al / dx);
int Nr = (int)ceil(Ar / dx);
NSpace = Nl + Nr;
Al = -Nl * dx;
Ar = Nr * dx;
Al = log(S0) - rho / omega * v0 + Al;
Ar = log(S0) - rho / omega * v0 + Ar;

int k, i, j;
int fv_up, fv_down;
double dt;
double **P_Old, **P_New;
double *A, *B, *C, *S;
double *beta_p, *u, *v, *Pr;
int N_fft;
int p;
int NN_fft;
int Nz;
double *mu;
double *mu_img;
double *uaux;
double *uaux_img;
double *somme;
double *somme_img;
double *vect_t;
double *vect_y;
double y0;
double *Price;
double factor;
double* tgrid;

/*Memory Allocation*/

//Tree construction for v
tree_v(T, v0, kappa, theta, omega, Ntime);
```

```
dt = T / (double)Ntime;

dx = (Ar - Al) / (NSpace);

const int Kmax = measure.Kmax;
const int Kmin = measure.Kmin;

int Ntp1 = Ntime + 1;
int NSp1 = NSpace + 1;
vect_t = (double*)malloc(Ntp1 * sizeof(double));
vect_y = (double*)malloc(NSp1 * sizeof(double));

for (int n = 0; n <= Ntime; n++)
vect_t[n] = n * dt;

P_Old = (double**)malloc(Ntp1 * sizeof(double*));
for (int i = 0; i <= Ntime; i++)
P_Old[i] = (double*)malloc(NSp1 * sizeof(double));

P_New = (double**)malloc(Ntp1 * sizeof(double*));
for (int i = 0; i <= Ntime; i++)
P_New[i] = (double*)malloc(NSp1 * sizeof(double));

//ADD
tgrid = (double *)malloc(Ntp1 * sizeof(double));
for (i = 0; i <= Ntime; i++) {
tgrid[i] = (double)i*dt;
}

//Terminal Condition
for (j = 0; j <= NSpace; j++)
vect_y[j] = Al + (double)j * dx;


factor = (1 - recovery_rate) *(exp(-default_intensity * tgrid[Ntime - 1]) - exp(
for (k = 0; k <= Ntime; k++)
for (int j = 0; j <= NSpace; j++)
{
P2[Ntime][j][k] = MAX(0, P[Ntime][j][k]) *factor;
P_Old[k][j] = P2[Ntime][j][k];
```

```
}

//some useful coefficients
/*matrix coefficients of the implicit part*/
A = (double*)malloc(NSp1 * sizeof(double));
B = (double*)malloc(NSp1 * sizeof(double));
C = (double*)malloc(NSp1 * sizeof(double));
S = (double*)malloc(NSp1 * sizeof(double));
Price = (double*)malloc(NSp1 * sizeof(double));
Pr = (double*)malloc(NSp1 * sizeof(double));
beta_p = (double*)malloc(NSp1 * sizeof(double));
u = (double*)malloc(NSp1 * sizeof(double));
v = (double*)malloc(NSp1 * sizeof(double));
int testa;
testa = 0;
N_fft = NSpace + Kmax - Kmin; //number of non-zero values of u involved in compu
  //zero-padding to obtain NN = N + Nz = 2^p
p = 1;
NN_fft = 2; //size of auxiliary vectors
while (NN_fft < N_fft)
{
p++;
NN_fft = 2 * NN_fft;
}
Nz = NN_fft - N_fft; // number of extra zeros

mu = (double*)malloc(NN_fft * sizeof(double));
mu_img = (double*)malloc(NN_fft * sizeof(double));
uaux = (double*)malloc(NN_fft * sizeof(double));
uaux_img = (double*)malloc(NN_fft * sizeof(double));
somme = (double*)malloc(NN_fft * sizeof(double));
somme_img = (double*)malloc(NN_fft * sizeof(double));

/*computation of the Fourier transform of nu*/
for (int i = 0; i < Kmax - Kmin + 1; i++)
{
mu[i] = (*measure.nu_array)[Kmax - Kmin - i];
mu_img[i] = 0;
}
for (int i = Kmax - Kmin + 1; i < NN_fft; i++)
{
```

```
mu[i] = 0;
mu_img[i] = 0;
}
fft1d(mu, mu_img, NN_fft, -1);

double discount = exp(-r * dt);

for (int n = Ntime; n >= 1; n--) //time iterations
{
// ADD for the second PIDE
factor = (1 - recovery_rate) *(exp(-default_intensity * tgrid[MAX(0, n - 2)]) -


for (k = 0; k <= n - 1; k++)
{
fv_up = f_up[n - 1][k];
fv_down = f_down[n - 1][k];

double z, vv;
z = r - divid - 0.5 * V[n - 1][k] - rho * kappa * (theta - V[n - 1][k]) / omega;
vv = 0.5 * V[n - 1][k] * (1. - SQR(rho));

double alpha, beta, gamma;
int PriceIndex;
double bound1, bound2;

if (call_or_put == 1)
{
bound1 = 0;
bound2 = compute_S(vect_y[NSpace], V[Ntime][k], omega, rho) * exp(-divid * (Ntim
}
else
{
bound1 = K * exp(-r * (Ntime - k) * dt) - compute_S(vect_y[0], V[Ntime][k], omeg
bound2 = 0;
}

alpha = (-vv * dt / SQR(dx) + z * dt / (2.*dx));
beta = 1 + vv * 2 * dt / SQR(dx);
gamma = (-vv * dt / SQR(dx) - z * dt / (2 * dx));
```

```
for (PriceIndex = 1; PriceIndex <= NSpace - 1; PriceIndex++)
{
A[PriceIndex] = alpha;
B[PriceIndex] = beta;
C[PriceIndex] = gamma;
}

/*Set Gauss*/
for (PriceIndex = NSpace - 2; PriceIndex >= 1; PriceIndex--)
B[PriceIndex] = B[PriceIndex] - C[PriceIndex] * A[PriceIndex + 1] / B[PriceIndex
for (PriceIndex = 1; PriceIndex < NSpace; PriceIndex++)
A[PriceIndex] = A[PriceIndex] / B[PriceIndex];
for (PriceIndex = 1; PriceIndex < NSpace - 1; PriceIndex++)
C[PriceIndex] = C[PriceIndex] / B[PriceIndex + 1];

//Mixture of terminal conditions
for (int i = 0; i <= NSpace; i++)
Price[i] = pu_f[n - 1][k] * P_Old[k + fv_up][i] + pd_f[n - 1][k] * P_Old[k + fv_

// /*calculation of the discretized integral using FFT*/
for (int i = 0; i < NSpace - 1; i++)
{
if ((Kmax + 1 + i < 0) || (Kmax + 1 + i >= NSpace))
{
uaux[i] = 0.;
uaux[i] = MAX(0., compute_S(Al + (double)(Kmax + 1 + i) * dx, V[n - 1][k], omega
}
else uaux[i] = Price[Kmax + 1 + i];
uaux_img[i] = 0;
}

for (int i = NSpace - 1; i < NSpace + Nz - 1; i++)
{
uaux[i] = 0; //zero-padding
uaux_img[i] = 0;
}
for (int i = NSpace + Nz - 1; i < NN_fft; i++)
{
if ((Kmin - NSpace - Nz + 1 + i < 0) || (Kmin - NSpace - Nz + 1 + i >= NSpace))
{
uaux[i] = 0.;
```

```
uaux[i] = MAX(0., compute_S(Al + (double)(Kmin - NSpace - Nz + 1 + i) * dx, V[n
}
else uaux[i] = Price[Kmin - NSpace - Nz + 1 + i];
uaux_img[i] = 0;
}

fft1d(uaux, uaux_img, NN_fft, -1);

for (int i = 0; i < NN_fft; i++)
{
somme[i] = mu[i] * uaux[i] - mu_img[i] * uaux_img[i];
somme_img[i] = mu_img[i] * uaux[i] + mu[i] * uaux_img[i];
}
fft1d(somme, somme_img, NN_fft, 1);

if (measure.alpha < 0)
{
S[0] = bound1 + dt * (somme[NN_fft - 1] - measure.alpha*(Price[1] - bound1) / dx
;
for (PriceIndex = 1; PriceIndex<NSpace; PriceIndex++)
S[PriceIndex] = Price[PriceIndex] + dt * (somme[PriceIndex - 1] - measure.alpha*

S[NSpace] = bound2 + dt * (somme[NN_fft - 1] - measure.alpha*(bound2 - Price[NSp
}
else
{
S[0] = bound1 + dt * (somme[NN_fft - 1] - measure.alpha*(Price[1] - bound1) / dx
;
for (PriceIndex = 1; PriceIndex<NSpace; PriceIndex++)
S[PriceIndex] = Price[PriceIndex] + dt * (somme[PriceIndex - 1] - measure.alpha*
S[NSpace] = bound2 + dt * (somme[NN_fft - 1] - measure.alpha*(bound2 - Price[NSp
}

/*Solve the system*/
for (PriceIndex = NSpace - 1; PriceIndex >= 1; PriceIndex--)
S[PriceIndex] = S[PriceIndex] - C[PriceIndex] * S[PriceIndex + 1];

Pr[1] = S[1] / B[1];


for (PriceIndex = 2; PriceIndex < NSpace; PriceIndex++)
```

```
Pr[PriceIndex] = S[PriceIndex] / B[PriceIndex] - A[PriceIndex] * Pr[PriceIndex -

P_New[k][0] = 0;
P_New[k][NSpace] = 0;


for (PriceIndex = 1; PriceIndex < NSpace; PriceIndex++)
{
P_New[k][PriceIndex] = discount * Pr[PriceIndex];
}

}//k iteration

 //Copy
for (j = 0; j <= NSpace; j++) { //for (j = 1; j < NSpace; j++)
for (k = 0; k <= n - 1; k++)
{
P_Old[k][j] = P_New[k][j] + factor * MAX(0, P[n - 1][j][k]);
P2[n - 1][j][k] = P_Old[k][j];
}
}
testa = testa + 1;
}//end of time iterations


 //s0 Index
int Index;
double sm, s, sp;

y0 = log(S0) - rho / omega * V[0][0];
Index = 0.;
while (vect_y[Index] < y0)
{
Index++;
}

Index--;
if (Index < 0) Index = 0;

sm = compute_S(vect_y[Index], V[0][0], omega, rho);
s = compute_S(y0, V[0][0], omega, rho);
```

```
sp = compute_S(vect_y[Index + 1], V[0][0], omega, rho);

//Price
*ptprice2 = P_New[0][Index] + (P_New[0][Index + 1] - P_New[0][Index]) * (s - sm)

//Delta
//*ptdelta = (P_New[0][Index + 1] - P_New[0][Index]) / (s - sm);


free(beta_p);
free(u);
free(v);

free(mu);
free(mu_img);
free(uaux);
free(uaux_img);
free(somme);
free(somme_img);

free(A);
free(B);
free(C);
free(S);
free(Pr);

free(vect_t);
free(vect_y);
free(Price);

for (i = 0; i <= Ntime; i++)
free(P_Old[i]);

free(P_Old);

for (i = 0; i <= Ntime; i++)
free(P_New[i]);

free(P_New);
free(tgrid);
```

```
return OK;

}


  int CVA_Bates(int am,double S0, NumFunc_1  *p, double T,double r, double divid
{
 int call_or_put;
 double K;
 double km=5;
 int dummy = 0;
 int dummy2= 0;
 double ptprice;

 if ((p->Compute) == &Call)
   call_or_put=1;
 else
   call_or_put=0;

 K=p->Par[0].Val.V_PDOUBLE;

  dummy  = FDHYBRIDTREE_Bates(call_or_put, am, K, T, S0, r, divid, v0, kappa, th
   dummy2 = FDHYBRIDTREE_Bates2(call_or_put, K, T, S0, r, divid, v0, kappa, thet


 *CVA=ptprice;

  return OK;

}

int CALC(FD_CVAMixedPDEBates)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r, divid;

  r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
  divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

  return CVA_Bates(ptOpt->EuOrAm.Val.V_BOOL,ptMod->S0.Val.V_PDOUBLE,
```

```
                    ptOpt->PayOff.Val.V_NUMFUNC_1,
                    ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                    r,
                    divid, ptMod->Sigma0.Val.V_PDOUBLE
                    , ptMod->MeanReversion.Val.V_PDOUBLE,
                    ptMod->LongRunVariance.Val.V_PDOUBLE,
                    ptMod->Sigma.Val.V_PDOUBLE,
                    ptMod->Rho.Val.V_PDOUBLE,  ptMod->Lambda.Val.V_PDOUBLE,
                                ptMod->Mean.Val.V_PDOUBLE,
   ptMod->Variance.Val.V_PDOUBLE,ptMod->Recovery.Val.V_DOUBLE,ptMod->Intensity.V

}

static int CHK_OPT(FD_CVAMixedPDEBates)(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)
{
  //int type_generator;
  if (Met->init == 0)
    {
      Met->init = 1;

  Met->HelpFilenameHint = "FD_CVAMixedPDEBates";
  Met->Par[0].Val.V_PINT = 100;
  Met->Par[1].Val.V_PDOUBLE = 0.01;
    }


  return OK;
}


PricingMethod MET(FD_CVAMixedPDEBates) =
{
```

```
  "FD_CVAMixedPDEBates",
  { { "Number of Time Steps", INT, {100}, ALLOW},{"Space Steps", PDOUBLE, {100},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_CVAMixedPDEBates),
   { {"CVA", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(FD_CVAMixedPDEBates),
  CHK_mc,
  MET(Init)
};
}
```