Help

```
extern "C" {
#include  "
href../../../mod/merhes1d_default/merhes1d_default_stdr/merhes1d_default_stdr_h_
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_tridiag_matrix.h"
#include "pnl/pnl_band_matrix.h"
#include "pnl/pnl_cdf.h"
}

extern "C" {
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2019+2) //The "#els
static int CHK_OPT(FD_CVAMixedPDEHeston)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(FD_CVAMixedPDEHeston)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else



static double **V, **P_old, **P_new, ***P, ***P2, ***vect_s;
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));

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);


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 ome
  {
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_Heston(int call_or_put, int am, double K, double tt, d
  {// First PDE
int i, j, k;
double stock;
int fv_up, fv_down;
double l;
double alpha, beta, gamma, alpha1, beta1, gamma1;
double dx;
double log_s0;
double discount;
double bound1, bound2;
double z, vv;
double dt;
double  *A, *B, *C, *A1, *B1, *C1, *Price, *S;
int Index, PriceIndex;
double sigma = 0.5;
double PRECISION_FDH = 1.0e-5;

//Tree construction for v
tree_v(tt, v0, kappa, theta, omega, Nt);
```

```
//Finite Difference algorithm
A = (double *)malloc((N + 1) * sizeof(double));
B = (double *)malloc((N + 1) * sizeof(double));
C = (double *)malloc((N + 1) * sizeof(double));
A1 = (double *)malloc((N + 1) * sizeof(double));
B1 = (double *)malloc((N + 1) * sizeof(double));
C1 = (double *)malloc((N + 1) * sizeof(double));

Price = (double *)malloc((N + 1) * sizeof(double));
S = (double *)malloc((N + 1) * sizeof(double));

dt = tt / (double)Nt;

l = sigma * sqrt(tt)*sqrt(log(1.0 / PRECISION_FDH)) + fabs((r_fisso - divid - 0.

dx = 2.0 * l / (double)N;
log_s0 = (log(s0) - rho / omega * V[0][0]);

//Mesh for y
for (j = 0; j <= N; j++)
  vect_y[j] = log_s0 - l + (double)j * dx;

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

/*Maturity conditions*/
for (k = 0; k <= Nt; k++)
  {
for (j = 0; j <= N; j++)
  {
stock = compute_S(vect_y[j], V[Nt][k], omega, rho);
if (call_or_put)
  P_old[j][k] = MAX(0, stock - K);
else P_old[j][k] = MAX(0, K - stock);
P[Nt][j][k] = P_old[j][k];
```

```
  }
  }

/*Rhs Factors*/
alpha1 = 0.;
beta1 = 1.;
gamma1 = 0.;

for (PriceIndex = 1; PriceIndex <= N - 1; PriceIndex++)
  {
A1[PriceIndex] = alpha1;
B1[PriceIndex] = beta1;
C1[PriceIndex] = gamma1;
  }

discount = exp(-r_fisso * dt);

/*Dynamic Programming*/
for (i = Nt - 1; i >= 0; i--)
  {
for (k = 0; k <= i; k++)
  {
z = (r_fisso - divid - 0.5 * V[i][k] - rho * kappa * (theta - V[i][k]) / omega);
vv = 0.5 * V[i][k] * (1. - SQR(rho));

fv_up = f_up[i][k];
fv_down = f_down[i][k];

if (call_or_put == 1)
  {
bound1 = 0;
bound2 = compute_S(vect_y[N], V[i][k], omega, rho) * exp(-divid * i * dt) - K *
  }
else
  {
bound1 = K * exp(-r_fisso * i * dt) - compute_S(vect_y[0], V[i][k], omega, rho)
bound2 = 0;
  }


//Fully Implicit
```

```
/*Lhs Factor of the fully implicit scheme*/
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 <= N - 1; PriceIndex++)
  {
A[PriceIndex] = alpha;
B[PriceIndex] = beta;
C[PriceIndex] = gamma;
  }

B[1] = beta + alpha;
B[N - 1] = beta + gamma;

/* B1[1]=beta1+alpha1; */
/* B1[N-1]=beta1+gamma1; */

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

//F_U

//Initialise
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
  {
Price[PriceIndex] = pu_f[i][k] * P_old[PriceIndex][k + fv_up] + pd_f[i][k] * P_o
  }

/*Set Rhs*/
S[1] = B1[1] * Price[1] + C1[1] * Price[2] + A1[1] * bound1 - alpha * bound1;
for (PriceIndex = 2; PriceIndex < N - 1; PriceIndex++)
  S[PriceIndex] = A1[PriceIndex] * Price[PriceIndex - 1] +
B1[PriceIndex] * Price[PriceIndex] +
C1[PriceIndex] * Price[PriceIndex + 1];
S[N - 1] = A1[N - 1] * Price[N - 2] + B1[N - 1] * Price[N - 1] + C1[N - 1] * bou
```

```
/*Solve the system*/
for (PriceIndex = N - 2; PriceIndex >= 1; PriceIndex--)
  S[PriceIndex] = S[PriceIndex] - C[PriceIndex] * S[PriceIndex + 1];

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

for (PriceIndex = 2; PriceIndex < N; PriceIndex++)
  Price[PriceIndex] = S[PriceIndex] / B[PriceIndex] - A[PriceIndex] * Price[Pric

for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
  {
P_new[PriceIndex][k] = discount * Price[PriceIndex];
  }
if (am)
  for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
  stock = compute_S(vect_y[PriceIndex], V[i][k], omega, rho);

  if (call_or_put)
P_new[PriceIndex][k] = MAX(P_new[PriceIndex][k], MAX(0, stock - K));
  else
P_new[PriceIndex][k] = MAX(P_new[PriceIndex][k], MAX(0, K - stock));
}
  }//end k

//Copy
for (j = 0; j <= N; j++)
  for (k = 0; k <= i; k++)
{
  P_old[j][k] = P_new[j][k];
  P[i][j][k] = P_old[j][k];
}

  }//end i

Index = (int)floor((double)N / 2.0);

/*Price*/
*ptprice = P[0][Index][0];
//*ptprice = P_old[Index][0];
```

```
/*Memory Disallocation*/
//free_memory(Nt, N);
free(A);
free(B);
free(C);
free(A1);
free(B1);
free(C1);
free(S);
free(Price);

return OK;

  }
static int FDHYBRIDTREE_Heston_2(int call_or_put, double K, double tt, double s0
{// Second PDE
int i, j, k;
int fv_up, fv_down;
double l;
double alpha, beta, gamma, alpha1, beta1, gamma1;
double dx;
double log_s0;
double discount;
double bound1, bound2;
double z, vv;
double dt;
double  *A, *B, *C, *A1, *B1, *C1, *Price, *S;
int Index, PriceIndex;
double sigma = 0.5;
double PRECISION_FDH = 1.0e-5;

double factor;
double* tgrid;

//Tree construction for v
//tree_v(tt, v0, kappa, theta, omega, Nt);

//Finite Difference algorithm
A = (double *)malloc((N + 1) * sizeof(double));
B = (double *)malloc((N + 1) * sizeof(double));
```

```
C = (double *)malloc((N + 1) * sizeof(double));
A1 = (double *)malloc((N + 1) * sizeof(double));
B1 = (double *)malloc((N + 1) * sizeof(double));
C1 = (double *)malloc((N + 1) * sizeof(double));

Price = (double *)malloc((N + 1) * sizeof(double));
S = (double *)malloc((N + 1) * sizeof(double));

dt = tt / (double)Nt;

l = sigma * sqrt(tt)*sqrt(log(1.0 / PRECISION_FDH)) + fabs((r_fisso - divid - 0.

dx = 2.0 * l / (double)N;
log_s0 = (log(s0) - rho / omega * V[0][0]);

//Mesh for y
for (j = 0; j <= N; j++)
vect_y[j] = log_s0 - l + (double)j * dx;

//Mesh of S
for (i = Nt; i >= 0; i--)
for (k = 0; k <= i; k++)
for (j = 0; j <= N; j++)
{
vect_s[i][j][k] = compute_S(vect_y[j], V[i][k], omega, rho);
}
//Mesh of t
tgrid = (double *)malloc((Nt + 1) * sizeof(double));
for (i = 0; i <= Nt; i++) {
tgrid[i] = (double)i*dt;
}

/*Maturity conditions*/
factor = (1 - recovery_rate) *(exp(-default_intensity * tgrid[N - 1]) - exp(-def

for (k = 0; k <= Nt; k++)
{
for (j = 0; j <= N; j++)
{

P2[Nt][j][k] = P[Nt][j][k]*factor;
```

```
P_old[j][k] = P2[Nt][j][k];
}
}

/*Rhs Factors*/
alpha1 = 0.;
beta1 = 1.;
gamma1 = 0.;

for (PriceIndex = 1; PriceIndex <= N - 1; PriceIndex++)
{
A1[PriceIndex] = alpha1;
B1[PriceIndex] = beta1;
C1[PriceIndex] = gamma1;
}

discount = exp(-r_fisso * dt);

/*Dynamic Programming*/
for (i = Nt - 1; i >= 0; i--)
{

factor = (1 - recovery_rate) *(exp(-default_intensity * tgrid[MAX(0, i-1)]) - ex

for (k = 0; k <= i; k++)
{
z = (r_fisso - divid - 0.5 * V[i][k] - rho * kappa * (theta - V[i][k]) / omega);
vv = 0.5 * V[i][k] * (1. - SQR(rho));

fv_up = f_up[i][k];
fv_down = f_down[i][k];

if (call_or_put == 1)
{
bound1 = 0;
bound2 = compute_S(vect_y[N], V[i][k], omega, rho) * exp(-divid * i * dt) - K *
}
else
{
bound1 = K * exp(-r_fisso * i * dt) - compute_S(vect_y[0], V[i][k], omega, rho)
bound2 = 0;
```

```
}


//Fully Implicit
/*Lhs Factor of the fully implicit scheme*/
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 <= N - 1; PriceIndex++)
{
A[PriceIndex] = alpha;
B[PriceIndex] = beta;
C[PriceIndex] = gamma;
}

B[1] = beta + alpha;
B[N - 1] = beta + gamma;


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

//F_U

//Initialise
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
Price[PriceIndex] = pu_f[i][k] * P_old[PriceIndex][k + fv_up] + pd_f[i][k] * P_o
}

/*Set Rhs*/
S[1] = B1[1] * Price[1] + C1[1] * Price[2] + A1[1] * bound1 - alpha * bound1;
for (PriceIndex = 2; PriceIndex < N - 1; PriceIndex++)
S[PriceIndex] = A1[PriceIndex] * Price[PriceIndex - 1] +
B1[PriceIndex] * Price[PriceIndex] +
```

```
C1[PriceIndex] * Price[PriceIndex + 1];
S[N - 1] = A1[N - 1] * Price[N - 2] + B1[N - 1] * Price[N - 1] + C1[N - 1] * bou

/*Solve the system*/
for (PriceIndex = N - 2; PriceIndex >= 1; PriceIndex--)
S[PriceIndex] = S[PriceIndex] - C[PriceIndex] * S[PriceIndex + 1];

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

for (PriceIndex = 2; PriceIndex < N; PriceIndex++)
Price[PriceIndex] = S[PriceIndex] / B[PriceIndex] - A[PriceIndex] * Price[PriceI

for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
{
P_new[PriceIndex][k] = discount * Price[PriceIndex];
}

}//end k

 //Copy
for (j = 0; j <= N; j++)
for (k = 0; k <= i; k++)
{
P_old[j][k] = P_new[j][k]+factor* P[i][j][k];
P2[i][j][k] = P_old[j][k];
}

}//end i

Index = (int)floor((double)N / 2.0);

/*Price*/
*ptprice2 = P2[0][Index][0];

/*Memory Disallocation*/
free(A);
free(B);
free(C);
free(A1);
free(B1);
free(C1);
```

```
free(S);
free(Price);
free(tgrid);

return OK;

}


////////////////////////////////////////////////////////////////////////////
//Compute CVA.

int CVA_Heston(int am,double S0, NumFunc_1  *p, double T, double K, double r, do
{
  int call_or_put;
  int dummy = 0;
double price;
double price2;

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

/*Memory Allocation*/
dummy = memory_allocation(Nt, N);

//Tree construction for v
dummy = tree_v(T, v0, kappa, theta, omega, Nt);

//Finite Difference Tree Algorithm
dummy = FDHYBRIDTREE_Heston(call_or_put, am, K, T, S0, r, divid, v0, kappa, thet

dummy = FDHYBRIDTREE_Heston_2(call_or_put,  K, T, S0, r, divid, v0, kappa, theta

*CVA = price2;

free_memory(Nt, N);

  return OK;
```

```
}

int CALC(FD_CVAMixedPDEHeston)(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_Heston(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,
   ptOpt->PayOff.Val.V_NUMFUNC_1->Par[0].Val.V_PDOUBLE,
                    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->Recovery.Val.V_DOUBLE,ptMod->

}

static int CHK_OPT(FD_CVAMixedPDEHeston)(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_CVAMixedPDEHeston";
```

```
  Met->Par[0].Val.V_PINT = 100;
  Met->Par[1].Val.V_PINT = 100;
    }


  return OK;
}


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