Help

```
#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"

#include "
href../../../common/enums_h_src.pdfenums.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2018+2) //The "#els
static int CHK_OPT(FD_Adi_HestonCVA)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(FD_Adi_HestonCVA)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

//Alfonsi code
static double psik (double t, double k)
{
    if (k==0.) return t;
    return (1-exp(-k*t))/k;
}

static double DiscLawMatch5(int generator)
{
  double u;

  u=pnl_rand_uni(generator);
    if (u<1./6.) return -sqrt(3);
    if (u<1./3.) return sqrt(3);
    return 0;
```

```
}

static double DiscLawMatch7(int generator)
{
    double u;

     u=pnl_rand_uni(generator);
    u=2.* u-1.; // THERE WAS STRANGE THING HERE !!! Local declaration with globa
    double res=sqrt(6);
    if (fabs(u)<((res-2)/(2*res))) res=sqrt(3+res);
    else res=sqrt(3-res);
    if (u<0) return -res;
    return res;

}

static double O3_1 (double t, double x, double a, double k, double sig)
{
    double aux;

    if (k==0) aux=t;
    else aux=(1-exp(-k*t))/k;
    return x*exp(-k*t)+(a-0.25*sig*sig)*aux;
}

static double O3_2 (double t, double x, double sig)
{
    double aux=MAX(sqrt(x)+0.5*sig*t,0);
    return SQR(aux);
}

static double O3_3 (double t, double x,double a, double k, double sig,int ordre)
{
    double aux=(a-0.25*sig*sig+k*x)*0.5*sig*sig;

    if (aux>0) return x+sqrt(aux)*t+k*sig*sig*0.125*t*t;
    if (aux<0) return x+sqrt(-aux)*t-k*sig*sig*0.125*t*t;
    return x;
}

static void Heston01(double *x1, double *x2, double *x3, double *x4, double dt,
```

```
              double a, double k, double sig, double mu, double rho, double Kseu
{
    double dx=0.,aux,ratio,p;
    double sig2=SQR(sig);
    double pp,ee;
    double u1, u2, u3;
    double s,res,dt2,dw2;
    int ordre;
    double rd=0;
    double u;

    if (flag_cir==1)
    {
        if (*x1>Kseuil*dt)
        {
            aux=exp(-k*0.5*dt);
            if (k==0.)
                u1=(a-SQR(0.5*sig))*dt*0.5;
            else   u1=(a-SQR(0.5*sig))*(1-aux)/k;
            dx=MAX(aux*SQR(sqrt(u1+aux*(*x1))+0.5*sig*dw)+u1-*x1,-*x1);
        }
        else
        {
            aux=exp(-k*dt);
            u1=(*x1)*aux+a*(1-aux)/k;
            ratio=(SQR(*x1*aux)+(2*a+sig*sig)*(((1-aux*aux)/(2*k))*a/k + (*x1-a/
            p=0.5*(1-sqrt(1-1/ratio));
            u=pnl_rand_uni(generator);
            if (u< p) dx=u1/(2*p)-*x1;
            else  dx=u1/(2*(1-p))-*x1;
        }
    }
    else  if (flag_cir==2)
    {
        if (*x1<Kseuil*dt)
        {
            ee=exp(-k*dt);
            if (k==0.) pp=dt;
            else pp=(1-ee)/k;
            u1=*x1*ee+a*pp;
            u2=u1*u1+sig2*pp*(0.5*a*pp+*x1*ee);
```

```
    u3=u1*u2+sig2*pp*(2**x1**x1*ee*ee+pp*(a+0.5*sig2)*(3**x1*ee+a*pp));
    s=(u3-u1*u2)/(u2-u1*u1);
    p=(u1*u3-u2*u2)/(u2-u1*u1);
    p=sqrt(s*s-4*p);
    u2=0.5*(s-p);
    u3=u2+p;
    u=pnl_rand_uni(generator);
    if (u<(u1-u2)/(u3-u2)) dx=u3-*x1;
    else dx=u2-*x1;
}
else
{
    if (a-0.25*sig2>0) ordre=1;
    else ordre=0;
    // On intègre 2

    if (k==0.) dt2=dt;
    else dt2=(exp(k*dt)-1)/k;
    dw2=sqrt(dt2/dt)*dw;

    // else ordre=-1;
    u=pnl_rand_uni(generator);
    rd=3*u;
    if (rd<1)
    {
        if (rd<0.5) s=-1;
        else s=1;
        if (ordre==1)
        {
            res=O3_3(s*dt2,*x1,a,0,sig,ordre);
            res=O3_2(dw2,res,sig);
            res=O3_1(dt2,res,a,0,sig);
        }
        else
        {
            res=O3_3(s*dt2,*x1,a,0,sig,ordre);
            res=O3_1(dt2,res,a,0,sig);
            res=O3_2(dw2,res,sig);
        }

        dx=exp(-k*dt)*res-*x1;
```

```
}
else
{
    if (rd<2)
    {
        if (rd-1<0.5) s=-1;
        else s=1;
        if (ordre==1)
        {
            res=O3_2(dw2,*x1,sig);
            res=O3_3(s*dt2,res,a,0,sig,ordre);
            res=O3_1(dt2,res,a,0,sig);
        }
        else
        {
            res=O3_1(dt2,*x1,a,0,sig);
            res=O3_3(s*dt2,res,a,0,sig,ordre);
            res=O3_2(dw2,res,sig);
        }
        dx=exp(-k*dt)*res-*x1;
    }
    else
        if (rd>=2.)
        {
            if (rd-2.<0.5) s=-1;
            else s=1;
            if (ordre==1)
            {
                res=O3_2(dw2,*x1,sig);
                res=O3_1(dt2,res,a,0,sig);
                res=O3_3(s*dt2,res,a,0,sig,ordre);
            }
            else
            {
                res=O3_1(dt2,*x1,a,0,sig);
                res=O3_2(dw2,res,sig);
                res=O3_3(s*dt2,res,a,0,sig,ordre);
            }

            dx=exp(-k*dt)*res-*x1;
        }
```

```
            }
        }
    }

    *x2=*x2+(*x1+0.5*dx)*dt;
    *x4=*x4+0.5*(*x3)*dt;
    *x3=*x3*exp((mu-rho*a/sig)*dt+rho*dx/sig+(rho*k/sig-0.5)*(*x1+0.5*dx)*dt);
    *x4=*x4+0.5*(*x3)*dt;
    *x1=*x1+dx;

    return;
}

static void Heston02 (double *x1, double *x3,double dw2, double rho)
{
    *x3=(*x3)*exp(sqrt((1-rho*rho)*(*x1))*dw2);
    return;
}

static void fct_Heston(double *x1, double *x2, double *x3, double *x4, double dt
{
  double u;
    u=pnl_rand_uni(generator);
    if ( u>0.5)
    {
        Heston02 ( x1, x3, dw2,  rho);
        Heston01 ( x1, x2, x3, x4, dt, dw, a,  k, sig,  mu,  rho, Kseuil,generat
    }
    else
    {
        Heston01 ( x1, x2, x3, x4, dt, dw, a,  k, sig,  mu,  rho, Kseuil,generat
        Heston02 ( x1, x3, dw2,  rho);
    }
    return;
}

//////////
// Functions to generate grids.
//////////
static double asinh1(double value)
{
```

```
    double returned;
    if(value>0)
        returned = log(value + sqrt(value * value + 1));
    else
        returned = -log(-value + sqrt(value * value + 1));
    return(returned);
}

static int lower_index(double *grid, int size, double value)
{
    double value_nearest;
    int index_nearest;
    int i;

    value_nearest = ABS(grid[0]-value);
    index_nearest = -1;

    for (i=0; i<size; i++)
    {
        if (ABS(grid[i]-value) <= value_nearest)
        {
            value_nearest = ABS(grid[i]-value);
            index_nearest = i;
        }
    }
    if (grid[index_nearest] > value)
    {
        return index_nearest-1;
    }
    else
    {
        return index_nearest;
    }
}

static void grid_generation_spot(double *sgrid, double Sleft, double Sright, dou
{
    int i;
    double Ximin;
    double Xiint;
    double Ximax;
```

```
    double deltaxi;

    Ximin = asinh1(-Sleft/Coeff_s);
    Xiint = (Sright-Sleft)/Coeff_s;
    Ximax = Xiint + asinh1((Smax-Sright)/Coeff_s);
    deltaxi = (Ximax-Ximin)/Ns;

    // Definition of uniform grid Xi.
    for (i=0; i<=Ns; i++)
    {
        sgrid[i] = Ximin + i * deltaxi;
    }

    // Definition of the spot grid with the uniform grid Xi.
    sgrid[0] = 0;
    for (i=1; i<=Ns; i++)
    {
        if (sgrid[i]<0)
        {
            sgrid[i] = Sleft+Coeff_s*sinh(sgrid[i]);
        }
        else
        {
            if (sgrid[i]<=Xiint)
            {
                sgrid[i] = Sleft+Coeff_s*sgrid[i];
            }
            else
            {
                sgrid[i] = Sright+Coeff_s*sinh(sgrid[i]-Xiint);
            }
        }
    }
}

static void grid_generation_variance(double *vgrid, double Vmax, int Nv, double
{
    int j;
    double deltaeta;

    deltaeta = (asinh1(Vmax/Coeff_v))/Nv;
```

```
    // Definition of uniform grid eta.
    for (j=0; j<=Nv; j++)
    {
        vgrid[j] = j * deltaeta;
    }
    // Definition of the volatility grid with the uniform grid eta.
    vgrid[0] = 0;
    for (j=1; j<=Nv; j++)
    {
        vgrid[j] = Coeff_v * sinh(vgrid[j]);
    }

        j = lower_index(vgrid, Nv, Center_v);
        vgrid[j] = Center_v;
}

//////////
// Functions to manage stencil and interpolate values.
//////////
static int stencil(int i, int j)
{
    if ((i== 0) && (j== 0)) return 0;
    if ((i==-1) && (j== 0)) return 1;
    if ((i== 1) && (j== 0)) return 2;
    if ((i== 0) && (j==-2)) return 3;
    if ((i== 0) && (j==-1)) return 4;
    if ((i== 0) && (j== 1)) return 5;
    if ((i== 0) && (j== 2)) return 6;
    if ((i==-1) && (j==-1)) return 7;
    if ((i== 1) && (j==-1)) return 8;
    if ((i==-1) && (j== 1)) return 9;
    if ((i== 1) && (j== 1)) return 10;
    /*
     0, 0 -> 0
     -1, 0 -> 1
     1, 0 -> 2
     0,-2 -> 3
     0,-1 -> 4
     0, 1 -> 5
     0, 2 -> 6
     -1,-1 -> 7
```

```
     1,-1 -> 8
     -1, 1 -> 9
     1, 1 -> 10

     6
     9 5 10
     1 0 2
     7 4 8
     3

     */
    printf("Error in stencil %d %d\ n",i,j);
    return -1;
}

static double interpolation(double griddown, double valuedown,
                            double gridup, double valueup, double gridunknown)
{
    return (valueup-valuedown)/(gridup-griddown) * (gridunknown - griddown) + va
}

static double double_interpolation(double value_sd_vd, double value_sd_vu,
                                   double value_su_vd, double value_su_vu,
                                   double grid_sd, double grid_su,
                                   double grid_vd, double grid_vu,
                                   double s, double v)
{
    double value_sd,value_su;
    value_sd = interpolation (grid_vd, value_sd_vd, grid_vu, value_sd_vu, v);
    value_su = interpolation (grid_vd, value_su_vd, grid_vu, value_su_vu, v);
    return interpolation (grid_sd, value_sd, grid_su, value_su, s);
}

static int it_exists_stencil(int i, int Ns, int j, int Nv, int stencil)
{
    // We use i from 0 to Ns, j from 0 to Nv.
    // We use points i-1 -> i+1 and j-2 -> j+2.

    /*
     0, 0 -> 0
     -1, 0 -> 1
```

```
 1, 0 -> 2
 0,-2 -> 3
 0,-1 -> 4
 0, 1 -> 5
 0, 2 -> 6
 -1,-1 -> 7
 1,-1 -> 8
 -1, 1 -> 9
 1, 1 -> 10

 6
 9 5 10
 1 0 2
 7 4 8
 3

 */

if ((i>0) && (i<Ns)&& (j>1) && (j<Nv-1)) // Strict interior domain for all s
{
    return 1;
}

if (i==0)
    if ((stencil== 1) || (stencil== 7) || (stencil== 9))
        return 0;

if (i==Ns)
    if ((stencil== 2) || (stencil== 8) || (stencil==10))
        return 0;

if (j==0)
    if ((stencil== 3) || (stencil== 4) || (stencil==7) || (stencil== 8))
        return 0;

if (j==1)
    if (stencil== 3)
        return 0;

if (j==Nv-1)
    if (stencil== 6)
```

```
            return 0;

    if (j==Nv)
        if ((stencil== 5) || (stencil== 6) || (stencil==9) || (stencil== 10))
            return 0;

    return 1;
}

static void point_of_stencil(int i, int j, int stencil, int* pi, int* pj)
{
    *pi=i;     *pj=j;
    if (stencil==0)  {   *pi=i;     *pj=j;  }
    if (stencil==1)  {   *pi=i-1;   *pj=j;  }
    if (stencil==2)  {   *pi=i+1;   *pj=j;  }
    if (stencil==3)  {   *pi=i;     *pj=j-2;}
    if (stencil==4)  {   *pi=i;     *pj=j-1;}
    if (stencil==5)  {   *pi=i;     *pj=j+1;}
    if (stencil==6)  {   *pi=i;     *pj=j+2;}
    if (stencil==7)  {   *pi=i-1;   *pj=j-1;}
    if (stencil==8)  {   *pi=i+1;   *pj=j-1;}
    if (stencil==9)  {   *pi=i-1;   *pj=j+1;}
    if (stencil==10) {   *pi=i+1;   *pj=j+1;}
}

//////////
// Functions to manage boundary conditions.
//////////
static double bc_spot_min(double S0, double *sgrid, int Ns, int is,
                          double V0, double sigma_v, double alpha_v, double beta
                          double R0, double rho_sv,
                          int call_or_put, double strike, double dividend,
                          double *tgrid, int Nt, int time_index)
{
    // dt U = sigma_v^2*V/2 dvv U + alpha_v (beta_v - V) dv U - r U
    // + 0 dss U + 0 ds U + 0 dsv U
    // Dirichlet condition
    return (call_or_put ? 0. : strike*exp(-R0*tgrid[Nt-time_index]));
}

static double bc_spot_max(double S0, double *sgrid, int Ns, int is,
```

```
                            double V0, double sigma_v, double alpha_v, double beta
                            double R0, double rho_sv,
                            int call_or_put, double strike, double dividend,
                            double *tgrid, int Nt, int time_index)
{
    // dt U = SSV/2 dss U + rS ds U + rho_sv sigma_v S V dsv U
    // + sigma_v^2*V/2 dvv U + alpha_v (beta_v - V) dv U - r U
    // Easy boundary condition -> Neumann = 1 ou 0.

    return (call_or_put ? 1. : 0.);
}

static double bc_var_max(double S0, double *sgrid, int Ns, int is,
                         double V0, double sigma_v, double alpha_v, double beta_
                         double R0, double rho_sv,
                         int call_or_put, double strike, double dividend,
                         double *tgrid, int Nt, int time_index)
{
    // Neumann = 0.
    return 0.;
}

//////////
// Functions to build matrix and solve systems.
//////////
static void build_all_matrix(double S0, double *sgrid, int Ns,
                             double V0, double sigma_v, double alpha_v, double b
                             double R0, double rho_sv,
                             int call_or_put, double strike, double dividend,
                             double *tgrid, int Nt, int time_index,
                             double ***MatrixA0, double ***MatrixA1, double ***M
                             double **G0, double **G1, double **G2)
{
    double actualspoint;
    double actualvpoint;
    double actualrpoint;

    double Dsi, Dsip1;
    double Dvjm1, Dvj, Dvjp1, Dvjp2;

    double convection_s, diffusion_s;
```

```
double convection_v, diffusion_v;
double order_0, mixted_sv;

double *cs;
double *ds;
double *cv;
double *dv;
double *msv;

// Coefficients
//    double salpha_im2, salpha_im1, salpha_i0;
double sbeta_im1, sbeta_i0, sbeta_ip1;
//    double sgamma_i0, sgamma_ip1, sgamma_ip2;
//    double sdelta_im1, sdelta_i0, sdelta_ip1;

//    double valpha_jm2, valpha_jm1, valpha_j0;
double vbeta_jm1, vbeta_j0, vbeta_jp1;
//    double vgamma_j0, vgamma_jp1, vgamma_jp2;
//    double vdelta_jm1, vdelta_j0, vdelta_jp1;

int i, j, st;

cs=(double*)malloc(11*sizeof(double));
ds=(double*)malloc(11*sizeof(double));
cv=(double*)malloc(11*sizeof(double));
dv=(double*)malloc(11*sizeof(double));
msv=(double*)malloc(11*sizeof(double));

double G_cs, G_ds, G_cv, G_dv, G_msv;

for(j=0;j<Nv+1;j++)
{
    for(i=1;i<Ns+1;i++)
    {
        for(st=0;st<11;st++)
        {
            cs[st]=0.;
            ds[st]=0.;
            cv[st]=0.;
            dv[st]=0.;
            msv[st]=0.;
```

```
}
G_cs=0.;
G_ds=0.;
G_cv=0.;
G_dv=0.;
G_msv=0.;

actualspoint = sgrid[i];
actualvpoint = vgrid[j];
actualrpoint = R0;

// Classical PDE
convection_s = (actualrpoint-dividend) * actualspoint;
diffusion_s = actualspoint * actualspoint * actualvpoint/2.0;
convection_v = alpha_v*(beta_v - actualvpoint); //convection_v = kap
diffusion_v = sigma_v * sigma_v * actualvpoint/2.0;
order_0 = -actualrpoint;
mixted_sv = rho_sv * sigma_v * actualspoint * actualvpoint;

////////////////////
// Diffusion and Convection S
////////////////////
{
    if (i==1) // S=Smin -> Dirichlet
    {
        Dsi = sgrid[i]-sgrid[i-1];
        Dsip1 = sgrid[i+1]-sgrid[i];
        //ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1));
        G_ds += 2.0/(Dsi*(Dsi+Dsip1)) * bc_spot_min(S0, sgrid, Ns, i
                                                    V0, sigma_v, alp
                                                    R0, rho_sv,
                                                    call_or_put, str
                                                    tgrid, Nt, time_
        ds[stencil(0,0)]=-2.0/(Dsi*Dsip1);
        ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1));

        //cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1));
        G_cs += -Dsip1/(Dsi*(Dsi+Dsip1)) * bc_spot_min(S0, sgrid, Ns
                                                       V0, sigma_v,
                                                       R0, rho_sv,
                                                       call_or_put,
```

```
                                                                tgrid, Nt, ti
    cs[stencil(0,0)]=(Dsip1-Dsi)/(Dsi*Dsip1);
    cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1));
}
else
{
    if (i==Ns) // S=Smax -> Neumann
    {
        Dsi = sgrid[i]-sgrid[i-1];
        Dsip1 = Dsi;
        ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1));
        ds[stencil(0,0)]=-2.0/(Dsi*Dsip1) + 2.0/(Dsip1*(Dsi+Dsip
        //ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1));
        G_ds = 2.0/(Dsip1*(Dsi+Dsip1)) * bc_spot_max(S0, sgrid,
                                                     V0, sigma_v
                                                     R0, rho_sv,
                                                     call_or_put
                                                     tgrid, Nt,

        cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1));
        cs[stencil(0,0)]=(Dsip1-Dsi)/(Dsi*Dsip1) + Dsi/(Dsip1*(D
        //cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1));
        G_cs += Dsi/(Dsip1*(Dsi+Dsip1)) * bc_spot_max(S0, sgrid,
                                                      V0, sigma_
                                                      R0, rho_sv
                                                      call_or_pu
                                                      tgrid, Nt,
    }
    else
    {
        Dsi = sgrid[i]-sgrid[i-1];
        Dsip1 = sgrid[i+1]-sgrid[i];
        ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1));
        ds[stencil(0,0)]=-2.0/(Dsi*Dsip1);
        ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1));

        cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1));
        cs[stencil(0,0)]=(Dsip1-Dsi)/(Dsi*Dsip1);
        cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1));
    }
}
```

```
}
////////////////////
// Diffusion and Convection V
////////////////////
{
    if (j==0) // V=Vmin
    {
        // V=Vmin -> Diffusion = 0 and no boundary conditions.

        // V=Vmin -> Forward in convection.
        Dvjp1 = vgrid[j+1]-vgrid[j];
        Dvjp2 = vgrid[j+2]-vgrid[j+1];
        cv[stencil(0,0)]=-(2.0*Dvjp1+Dvjp2)/(Dvjp1*(Dvjp1+Dvjp2));
        cv[stencil(0,1)]=(Dvjp1+Dvjp2)/(Dvjp1*Dvjp2);
        cv[stencil(0,2)]=-Dvjp1/(Dvjp2*(Dvjp1+Dvjp2));
    }
    else
    {
        if (j==Nv) // V=Vmax
        {
            // V=Vmax -> Neumann for diffusion.
            Dvjm1 = vgrid[j-1]-vgrid[j-2];
            Dvj = vgrid[j]-vgrid[j-1];
            Dvjp1 = Dvj;
            dv[stencil(0,-1)]=2.0/(Dvj*(Dvj+Dvjp1));
            dv[stencil(0,0)]=-2.0/(Dvj*Dvjp1) + 2.0/(Dvjp1*(Dvj+Dvjp
            //dv[stencil(0,1)]=2.0/(Dvjp1*(Dvj+Dvjp1));
            G_dv += 2.0/(Dvjp1*(Dvj+Dvjp1)) * bc_var_max(S0, sgrid,
                                                         V0, sigma_v
                                                         R0, rho_sv,
                                                         call_or_put
                                                         tgrid, Nt,

            // V=Vmax -> Backward for convection.
            cv[stencil(0,0)]=(Dvjm1+2.0*Dvj)/(Dvj*(Dvjm1+Dvj));
            cv[stencil(0,-1)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj);
            cv[stencil(0,-2)]=Dvj/(Dvjm1*(Dvjm1+Dvj));
        }
        else
        {
            // If 1 <= j <= Nv-1 -> use central scheme for diffusion
```

```
                Dvj = vgrid[j]-vgrid[j-1];
                Dvjp1 = vgrid[j+1]-vgrid[j];

                dv[stencil(0,-1)]=2.0/(Dvj*(Dvj+Dvjp1));
                dv[stencil(0,0)]=-2.0/(Dvj*Dvjp1);
                dv[stencil(0,1)]=2.0/(Dvjp1*(Dvj+Dvjp1));

                // If 1 <= j <= Nv-1 -> scheme for convection depends on
                if ((convection_v<0) && (j>1)) // var > beta_v and j>1 -
                {
                    Dvjm1 = vgrid[j-1]-vgrid[j-2];

                    cv[stencil(0,-2)]=Dvj/(Dvjm1*(Dvjm1+Dvj));
                    cv[stencil(0,-1)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj);
                    cv[stencil(0,0)]=(Dvjm1+2.0*Dvj)/(Dvj*(Dvjm1+Dvj));
                }
                else // var <= beta_v or j==1 -> Central.
                {
                    Dvjm1 = vgrid[j]-vgrid[j-1];

                    cv[stencil(0,-1)]=-Dvjp1/(Dvj*(Dvj+Dvjp1));
                    cv[stencil(0,0)]=(Dvjp1-Dvj)/(Dvj*Dvjp1);
                    cv[stencil(0,1)]=Dvj/(Dvjp1*(Dvj+Dvjp1));
                }
            }
        }
    }

    // Mixted SV
    {
        // Important /!\
        // If S=0 or V=0 ->> misted_sv = 0. Nothing to do in all this re

        // Boundary condition elsewhere :
        // if V=Vmax, we impose Neumann on V independent of S
        // ->> no problem at S=Smin since mixted_sv = 0.
        // ->> no problem at S=Smax since also Neumann on S independent
        // if S=Smax, we impose Neumann on S independent of V
        // ->> no problem at V=Vmin since mixted_sv = 0.
        // ->> no problem at V=Vmax since also Neumann on V independent
```

```
if ((1<i) && (i<Ns) && (0<j) && (j<Nv)) // Strict interior 1<i<N
{
    Dsi = sgrid[i]-sgrid[i-1];
    Dsip1 = sgrid[i+1]-sgrid[i];
    Dvj = vgrid[j]-vgrid[j-1];
    Dvjp1 = vgrid[j+1]-vgrid[j];

    sbeta_im1 = -Dsip1/(Dsi*(Dsi+Dsip1));
    sbeta_i0 = (Dsip1-Dsi)/(Dsi*Dsip1);
    sbeta_ip1 = Dsi/(Dsip1*(Dsi+Dsip1));
    vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
    vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
    vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));

    msv[stencil(0,0)] = sbeta_i0 * vbeta_j0;
    msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0;
    msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0;
    msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1;
    msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1;
    msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1;
    msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1;
    msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1;
    msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1;
}
else // i==1 or i==Ns or j==0 or j==Nv
{
    if (j==Nv) // V=Vmax
    {
        // Three cases :
        // i==1 -> Condition on Vmin
        if (i==1) // S=Smin -> Dirichlet
        {
            Dsip1 = sgrid[i+1]-sgrid[i];
            Dsi = Dsip1;
            Dvj = vgrid[j]-vgrid[j-1];
            Dvjp1 = Dvj;

            sbeta_im1 = -Dsip1/(Dsi*(Dsi+Dsip1));
            sbeta_i0 = (Dsip1-Dsi)/(Dsi*Dsip1);
            sbeta_ip1 = Dsi/(Dsip1*(Dsi+Dsip1));
            vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
```

```
vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));

msv[stencil(0,0)] = sbeta_i0 * vbeta_j0;
//msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0;
G_msv += sbeta_im1 * vbeta_j0 * bc_spot_min(S0, sgri
                                            V0, sigm
                                            R0, rho_
                                            call_or_
                                            tgrid, N
msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0;
msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1;
//msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1;
G_msv += sbeta_i0 * vbeta_jp1 * bc_var_max(S0, sgrid
                                           V0, sigma
                                           R0, rho_s
                                           call_or_p
                                           tgrid, Nt
//msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1;
G_msv += sbeta_im1 * vbeta_jm1 * bc_spot_min(S0, sgr
                                             V0, sig
                                             R0, rho
                                             call_or
                                             tgrid,
msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1;
//msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1;
G_msv += sbeta_im1 * vbeta_jp1 * bc_spot_min(S0, sgr
                                             V0, sig
                                             R0, rho
                                             call_or
                                             tgrid,
//msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1;
G_msv += sbeta_ip1 * vbeta_jp1 * bc_var_max(S0, sgri
                                            V0, sigm
                                            R0, rho_
                                            call_or_
                                            tgrid, N

msv[stencil(0,0)] = sbeta_i0 * vbeta_j0
//+ 1. * sbeta_im1 * vbeta_j0 // Dirichlet
+ 1. * sbeta_i0 * vbeta_jp1 // Neumann
```

```
    //+ 1. * sbeta_im1 * vbeta_jm1 // Dirichlet
    //+ 1. * sbeta_im1 * vbeta_jp1 // Dirichlet
    + 1. * sbeta_ip1 * vbeta_jp1; // Neumann
}
// 1<i<Ns -> Condition on V=Vmax.
// i==Ns -> Condition on V=Vmax compatible with conditio
if ((1<i) && (i<Ns))
{
    Dsi = sgrid[i]-sgrid[i-1];
    Dsip1 = sgrid[i+1]-sgrid[i];
    Dvj = vgrid[j]-vgrid[j-1];
    Dvjp1 = Dvj;

    sbeta_im1 = -Dsip1/(Dsi*(Dsi+Dsip1));
    sbeta_i0 = (Dsip1-Dsi)/(Dsi*Dsip1);
    sbeta_ip1 = Dsi/(Dsip1*(Dsi+Dsip1));
    vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
    vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
    vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));


    msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0;
    msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0;
    msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1;
    //msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1;
    G_msv += sbeta_i0 * vbeta_jp1 * bc_var_max(S0, sgrid
                                               V0, sigma
                                               R0, rho_s
                                               call_or_p
                                               tgrid, Nt
    msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1;
    msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1;
    //msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1;
    G_msv += sbeta_im1 * vbeta_jp1 * bc_var_max(S0, sgri
                                                V0, sigm
                                                R0, rho_
                                                call_or_
                                                tgrid, N
    //msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1;
    G_msv += sbeta_ip1 * vbeta_jp1 * bc_var_max(S0, sgri
                                               V0, sigm
```

```
                                                        R0, rho_
                                                        call_or_
                                                        tgrid, N
        msv[stencil(0,0)] = sbeta_i0 * vbeta_j0
        + 1. * sbeta_im1 * vbeta_jp1 // Neumann
        + 1. * sbeta_i0 * vbeta_jp1 // Neumann
        + 1. * sbeta_ip1 * vbeta_jp1; // Neumann
    }
} // End of V=Vmax -> we do not have done i==Ns in j==Nv.
if (i==Ns) // S=Smax
{
    // Three cases :
    // j==0 -> mixted_sv = 0 -> Nothing to do.
    if (j==0)
    {
        Dsi = sgrid[i]-sgrid[i-1];
        Dsip1 = Dsi;
        Dvjp1 = vgrid[j+1]-vgrid[j];
        Dvj = Dvjp1;

        sbeta_im1 = -Dsip1/(Dsi*(Dsi+Dsip1));
        sbeta_i0 = (Dsip1-Dsi)/(Dsi*Dsip1);
        sbeta_ip1 = Dsi/(Dsip1*(Dsi+Dsip1));
        vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
        vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
        vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));

        msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0;
        //msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0;
        G_msv += sbeta_ip1 * vbeta_j0 * bc_spot_max(S0, sgri
                                                    V0, sigm
                                                    R0, rho_
                                                    call_or_
                                                    tgrid, N
        //msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1;
        G_msv += sbeta_i0 * vbeta_jm1 * bc_spot_max(S0, sgri
                                                    V0, sigm
                                                    R0, rho_
                                                    call_or_
                                                    tgrid, N
        msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1;
```

```
    //msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1;
    G_msv += sbeta_im1 * vbeta_jm1 * bc_spot_max(S0, sgr
                                                 V0, sig
                                                 R0, rho
                                                 call_or
                                                 tgrid,
    //msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1;
    G_msv += sbeta_ip1 * vbeta_jm1 * bc_spot_max(S0, sgr
                                                 V0, sig
                                                 R0, rho
                                                 call_or
                                                 tgrid,
    msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1;
    //msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1;
    G_msv += sbeta_ip1 * vbeta_jp1 * bc_spot_max(S0, sgr
                                                 V0, sig
                                                 R0, rho
                                                 call_or
                                                 tgrid,

    msv[stencil(0,0)] = sbeta_i0 * vbeta_j0
    + 1. * sbeta_ip1 * vbeta_jm1 // Neumann
    + 1. * sbeta_i0 * vbeta_jm1 // Neumann
    + 1. * sbeta_im1 * vbeta_jm1 // Neumann
    + 1. * sbeta_ip1 * vbeta_j0 // Neumann
    + 1. * sbeta_ip1 * vbeta_jp1; // Neumann
}
// 0<j<Nv -> Condition on S=Smax.
// j==Nv -> Condition on S=Smax compatible with conditio
if ((0<j) && (j<Nv))
{
    Dsi = sgrid[i]-sgrid[i-1];
    Dsip1 = Dsi;
    Dvj = vgrid[j]-vgrid[j-1];
    Dvjp1 = Dvj;

    sbeta_im1 = -Dsip1/(Dsi*(Dsi+Dsip1));
    sbeta_i0 = (Dsip1-Dsi)/(Dsi*Dsip1);
    sbeta_ip1 = Dsi/(Dsip1*(Dsi+Dsip1));
    vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
    vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
```

```
            vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));


            msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0;
            //msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0;
            G_msv += sbeta_ip1 * vbeta_j0 * bc_spot_max(S0, sgri
                                                        V0, sigm
                                                        R0, rho_
                                                        call_or_
                                                        tgrid, N
            msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1;
            msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1;
            msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1;
            //msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1;
            G_msv += sbeta_ip1 * vbeta_jm1 * bc_spot_max(S0, sgr
                                                         V0, sig
                                                         R0, rho
                                                         call_or
                                                         tgrid,
            msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1;
            //msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1;
            G_msv += sbeta_ip1 * vbeta_jp1 * bc_spot_max(S0, sgr
                                                         V0, sig
                                                         R0, rho
                                                         call_or
                                                         tgrid,
            msv[stencil(0,0)] = sbeta_i0 * vbeta_j0
            + 1. * sbeta_ip1 * vbeta_jm1 // Neumann
            + 1. * sbeta_ip1 * vbeta_j0 // Neumann
            + 1. * sbeta_ip1 * vbeta_jp1; // Neumann
        }
    } // End of S=Smax -> we do not have done j==Nv in i==Ns. Do
    if ((i==Ns) && (j==Nv)) // Corner ! The two Neumann conditio
    {
        Dsi = sgrid[i]-sgrid[i-1];
        Dsip1 = Dsi;
        Dvj = vgrid[j]-vgrid[j-1];
        Dvjp1 = Dvj;

        sbeta_im1 = -Dsip1/(Dsi*(Dsi+Dsip1));
        sbeta_i0 = (Dsip1-Dsi)/(Dsi*Dsip1);
```

```
sbeta_ip1 = Dsi/(Dsip1*(Dsi+Dsip1));
vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));


msv[stencil(-1,0)] = sbeta_im1 * vbeta_j0;
//msv[stencil(1,0)] = sbeta_ip1 * vbeta_j0;
G_msv += sbeta_ip1 * vbeta_j0 * bc_spot_max(S0, sgrid, N
                                            V0, sigma_v,
                                            R0, rho_sv,
                                            call_or_put,
                                            tgrid, Nt, t
msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1;
//msv[stencil(0,1)] = sbeta_i0 * vbeta_jp1;
G_msv += sbeta_i0 * vbeta_jp1 * bc_var_max(S0, sgrid, Ns
                                           V0, sigma_v,
                                           R0, rho_sv,
                                           call_or_put,
                                           tgrid, Nt, ti
msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1;
//msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1;
G_msv += sbeta_ip1 * vbeta_jm1 * bc_spot_max(S0, sgrid,
                                             V0, sigma_v
                                             R0, rho_sv,
                                             call_or_put
                                             tgrid, Nt,
//msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1;
G_msv += sbeta_im1 * vbeta_jp1 * bc_var_max(S0, sgrid, N
                                            V0, sigma_v,
                                            R0, rho_sv,
                                            call_or_put,
                                            tgrid, Nt, t
//msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1; !!!! Here C
G_msv += sbeta_ip1 * vbeta_jp1 * bc_spot_max(S0, sgrid,
                                             V0, sigma_v
                                             R0, rho_sv,
                                             call_or_put
                                             tgrid, Nt,
msv[stencil(0,0)] = sbeta_i0 * vbeta_j0
+ 1. * sbeta_ip1 * vbeta_j0 // Neumann
```

```
                        + 1. * sbeta_i0 * vbeta_jp1 // Neumann
                        + 1. * sbeta_ip1 * vbeta_jm1 // Neumann
                        + 1. * sbeta_im1 * vbeta_jp1 // Neumann
                        + 1. * sbeta_ip1 * vbeta_jp1; // Neumann
                    }
                }
            }

            // Central point for order_0
            MatrixA0[i][j][0] = mixted_sv * msv[0];
            MatrixA1[i][j][0] = order_0 * 0.5 + convection_s * cs[0] + diffusion
            MatrixA2[i][j][0] = order_0 * 0.5 + convection_v * cv[0] + diffusion

            for (st=1;st<11;st++)
            {
                MatrixA0[i][j][st] = mixted_sv * msv[st];
                MatrixA1[i][j][st] = convection_s * cs[st] + diffusion_s * ds[st
                MatrixA2[i][j][st] = convection_v * cv[st] + diffusion_v * dv[st
            }

            // Second members
            G0[i][j] = mixted_sv * G_msv;
            G1[i][j] = convection_s * G_cs + diffusion_s * G_ds;
            G2[i][j] = convection_v * G_cv + diffusion_v * G_dv;
        }
    }

    free(msv);
    free(dv);
    free(cv);
    free(ds);
    free(cs);
}

static void compute_explicit_syslin_all_matrix(double coeff,
                                               double *sgrid, int Ns, double *vg
                                               double ***MatrixA0, double ***Mat
                                               double **G0nm1, double **G1nm1, d
                                               double **Unm1, double **Y0)
{
```

```
    int i, j, st;
    double val=0.;
    int istencil, jstencil;

    for (i=1; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            val = Unm1[i][j];
            for (st=0; st<11; st++)
            {
                if (it_exists_stencil(i, Ns, j, Nv, st))
                {
                    // If the point exists.
                    point_of_stencil(i, j, st, &istencil, &jstencil);
                    val += coeff * MatrixA0[i][j][st] * Unm1[istencil][jstencil]
                    val += coeff * MatrixA1[i][j][st] * Unm1[istencil][jstencil]
                    val += coeff * MatrixA2[i][j][st] * Unm1[istencil][jstencil]
                }
            }
            val += coeff * G0nm1[i][j];
            val += coeff * G1nm1[i][j];
            val += coeff * G2nm1[i][j];
            Y0[i][j] = val;
        }
    }
}

static void computation_explicit_syslin_spot_matrix(double coeff,
                                                    double *sgrid, int Ns, doubl
                                                    double ***MatrixA0, double *
                                                    double **G0nm1, double **G1n
                                                    double **Unm1, double **Y0,
{
    int i, j, st;
    double val;
    int istencil, jstencil;

    val=0.;

    for (j=0; j<Nv+1; j++)
```

```
    {
        for (i=1; i<Ns+1; i++)
        {
            val = Y0[i][j];
            for (st=0; st<11; st++)
            {
                if (it_exists_stencil(i, Ns, j, Nv, st))
                {
                    // If the point exists.
                    point_of_stencil(i, j, st, &istencil, &jstencil);
                    val += -coeff * MatrixA1[i][j][st] * Unm1[istencil][jstencil
                }
            }
            val += -coeff * G1nm1[i][j];
            Sortie[i][j] = val;
        }
    }
}

static void computation_implicit_syslin_spot_matrix(double coeff,
                                                    double *sgrid, int Ns, doubl
                                                    double ***MatrixA0, double *
                                                    double **G0, double **G1, do
                                                    double **rhs, double **lhs)
{
    int i, j;

    // Only points from i=1 to i=Ns.
    // Only points from j=0 to j=Nv.
    PnlTridiagMat *Identity_Matrix;
    PnlTridiagMat *Working_Matrix;
    PnlVect *Entree;
    PnlVect *Sortie;

    Identity_Matrix = pnl_tridiag_mat_create_from_two_double(Ns, 1.0, 0.0); // I
    Working_Matrix = pnl_tridiag_mat_create(Ns);
    Entree = pnl_vect_create(Ns);
    Sortie = pnl_vect_create(Ns);

    for (j=0; j<Nv+1; j++)
    {
```

```
        // Build the matrix

        //pnl_tridiag_mat_set (Working_Matrix, 0, -1, MatrixA1[1][j][1]);
        pnl_tridiag_mat_set (Working_Matrix, 0,  0, MatrixA1[1][j][0]);
        pnl_tridiag_mat_set (Working_Matrix, 0,  1, MatrixA1[1][j][2]);
        for (i=1; i<Ns-1; i++)
        {
            pnl_tridiag_mat_set (Working_Matrix, i, -1, MatrixA1[i+1][j][1]);
            pnl_tridiag_mat_set (Working_Matrix, i,  0, MatrixA1[i+1][j][0]);
            pnl_tridiag_mat_set (Working_Matrix, i,  1, MatrixA1[i+1][j][2]);
        }
        pnl_tridiag_mat_set (Working_Matrix, Ns-1, -1, MatrixA1[Ns][j][1]);
        pnl_tridiag_mat_set (Working_Matrix, Ns-1,  0, MatrixA1[Ns][j][0]);
        //pnl_tridiag_mat_set (Working_Matrix, Ns-1,  1, MatrixA1[Ns][j][2]);

        // Multiplication by -coeff.
        pnl_tridiag_mat_mult_double(Working_Matrix, -coeff);
        // Sum with the identity matrix. Result is in the first argument.
        pnl_tridiag_mat_plus_tridiag_mat(Working_Matrix, Identity_Matrix);

        // Build the vectors.
        for (i=0; i<Ns; i++)
        {
            pnl_vect_set (Entree, i, rhs[i+1][j] + coeff * G1[i+1][j]);
        }

        pnl_tridiag_mat_syslin(Sortie, Working_Matrix, Entree);

        for (i=0; i<Ns; i++)
        {
            lhs[i+1][j] = pnl_vect_get (Sortie, i);
        }
    }

    pnl_vect_free(&Sortie);
    pnl_vect_free(&Entree);
    pnl_tridiag_mat_free(&Working_Matrix);
    pnl_tridiag_mat_free(&Identity_Matrix);
}

static void computation_explicit_syslin_var_matrix(double coeff,
```

```
                                                    double *sgrid, int Ns, double
                                                    double ***MatrixA0, double **
                                                    double **G0nm1, double **G1nm
                                                    double **Unm1, double **Y1, d
{
    int i, j, st;
    double val;
    int istencil, jstencil;

    val =0.;

    for (i=1; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            val = Y1[i][j];
            for (st=0; st<11; st++)
            {
                if (it_exists_stencil(i, Ns, j, Nv, st))
                {
                    // If the point exists.
                    point_of_stencil(i, j, st, &istencil, &jstencil);
                    val += -coeff * MatrixA2[i][j][st] * Unm1[istencil][jstencil
                }
            }
            val += -coeff * G2nm1[i][j];
            Sortie[i][j] = val;
        }
    }
}

static void computation_implicit_syslin_var_matrix(double coeff,
                                                    double *sgrid, int Ns, double
                                                    double ***MatrixA0, double **
                                                    double **G0, double **G1, dou
                                                    double **rhs, double **lhs)
{

    int i, j;

    // Only points from i=1 to i=Ns.
```

```
// Only points from j=0 to j=Nv.
// Pentadiagonal matrix.
PnlBandMat *Working_Matrix;
PnlVect *Entree;
PnlVect *Sortie;

Entree = pnl_vect_create(Nv+1);
Sortie = pnl_vect_create(Nv+1);

for (i=1; i<Ns+1; i++)
{
    Working_Matrix = pnl_band_mat_create(Nv+1,Nv+1,2,2);
    // Build the matrix

    //pnl_band_mat_set (Working_Matrix, 0, 0-2, MatrixA2[i][0][3]);
    //pnl_band_mat_set (Working_Matrix, 0, 0-1, MatrixA2[i][0][4]);
    pnl_band_mat_set (Working_Matrix, 0, 0+0, MatrixA2[i][0][0]);
    pnl_band_mat_set (Working_Matrix, 0, 0+1, MatrixA2[i][0][5]);
    pnl_band_mat_set (Working_Matrix, 0, 0+2, MatrixA2[i][0][6]);
    //pnl_band_mat_set (Working_Matrix, 1, 1-2, MatrixA2[i][1][3]);
    pnl_band_mat_set (Working_Matrix, 1, 1-1, MatrixA2[i][1][4]);
    pnl_band_mat_set (Working_Matrix, 1, 1+0, MatrixA2[i][1][0]);
    pnl_band_mat_set (Working_Matrix, 1, 1+1, MatrixA2[i][1][5]);
    pnl_band_mat_set (Working_Matrix, 1, 1+2, MatrixA2[i][1][6]);
    for (j=2; j<Nv-1; j++)
    {
        pnl_band_mat_set (Working_Matrix, j, j-2, MatrixA2[i][j][3]);
        pnl_band_mat_set (Working_Matrix, j, j-1, MatrixA2[i][j][4]);
        pnl_band_mat_set (Working_Matrix, j, j+0, MatrixA2[i][j][0]);
        pnl_band_mat_set (Working_Matrix, j, j+1, MatrixA2[i][j][5]);
        pnl_band_mat_set (Working_Matrix, j, j+2, MatrixA2[i][j][6]);
    }
    pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1-2, MatrixA2[i][Nv-1][3]);
    pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1-1, MatrixA2[i][Nv-1][4]);
    pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1+0, MatrixA2[i][Nv-1][0]);
    pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1+1, MatrixA2[i][Nv-1][5]);
    //pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1+2, MatrixA2[i][Nv-1][6]);
    pnl_band_mat_set (Working_Matrix, Nv, Nv-2, MatrixA2[i][Nv][3]);
    pnl_band_mat_set (Working_Matrix, Nv, Nv-1, MatrixA2[i][Nv][4]);
    pnl_band_mat_set (Working_Matrix, Nv, Nv+0, MatrixA2[i][Nv][0]);
    //pnl_band_mat_set (Working_Matrix, Nv, Nv+1, MatrixA2[i][Nv][5]);
```

```
        //pnl_band_mat_set (Working_Matrix, Nv, Nv+2, MatrixA2[i][Nv][6]);

        // Multiplication by -coeff.
        pnl_band_mat_mult_double(Working_Matrix, -coeff);
        // Sum with the identity matrix.
        for (j=0; j<Nv+1; j++)
        {
            pnl_band_mat_set(Working_Matrix, j, j, 1. + pnl_band_mat_get(Working
        }

        // Build the vectors.
        for (j=0; j<Nv+1; j++)
        {
            pnl_vect_set (Entree, j, rhs[i][j] + coeff * G2[i][j]);
        }

        pnl_band_mat_syslin(Sortie, Working_Matrix, Entree);

        for (j=0; j<Nv+1; j++)
        {
            lhs[i][j] = pnl_vect_get (Sortie, j);
        }
        pnl_band_mat_free(&Working_Matrix);
    }

    pnl_vect_free(&Sortie);
    pnl_vect_free(&Entree);
}

////////////////////////////////////////////////////////////////////////////
//Compute all prices.
////////////////////////////////////////////////////////////////////////////
static void compute_all_prices(double S0, double *sgrid, int Ns,
                               double V0, double sigma_v, double alpha_v, double
                               double *vgrid, int Nv,
                               double R0, double rho_sv,
                               int call_or_put, double strike, double dividend,
                               double *tgrid, int Nt,
                               double theta,  int scheme,
                               double ***U_time)
{
```

```
// Variables for loops.
int i, j, st, time_index;
double deltat;

// 2-vectors
double **Unm1;
double **Utmp;
double **Y0;
double **Y1;
//double **Y2;
//double **Y3;
double **G0nm1;
double **G1nm1;
double **G2nm1;
//double **G3nm1;
//double **G0n;
double **G1n;
double **G2n;
//double **G3n;

// Matrix (=3-vectors)
double ***MatrixA0nm1;
double ***MatrixA1nm1;
double ***MatrixA2nm1;
//double ***MatrixA3nm1;
//double ***MatrixA0n;
double ***MatrixA1n;
double ***MatrixA2n;
//double ***MatrixA3n;

// Memory allocations.
{
    // Memory allocation of 2-vectors.
    Unm1 = (double**) malloc((Ns+1) * sizeof(double*));
    Utmp = (double**) malloc((Ns+1) * sizeof(double*));
    Y0 = (double**) malloc((Ns+1) * sizeof(double*));
    Y1 = (double**) malloc((Ns+1) * sizeof(double*));
    //Y2 = (double**) malloc((Ns+1) * sizeof(double*));
    //Y3 = (double**) malloc((Ns+1) * sizeof(double*));
    G0nm1 = (double**) malloc((Ns+1) * sizeof(double*));
```

```
G1nm1 = (double**) malloc((Ns+1) * sizeof(double*));
G2nm1 = (double**) malloc((Ns+1) * sizeof(double*));
//G3nm1 = (double**) malloc((Ns+1) * sizeof(double*));
//G0n = (double**) malloc((Ns+1) * sizeof(double*));
G1n = (double**) malloc((Ns+1) * sizeof(double*));
G2n = (double**) malloc((Ns+1) * sizeof(double*));
//G3n = (double**) malloc((Ns+1) * sizeof(double*));
for (i = 0; i < Ns+1; i++)
{
    Unm1[i] = (double*) malloc((Nv+1) * sizeof(double));
    Utmp[i] = (double*) malloc((Nv+1) * sizeof(double));
    Y0[i] = (double*) malloc((Nv+1) * sizeof(double));
    Y1[i] = (double*) malloc((Nv+1) * sizeof(double));
    //Y2[i] = (double*) malloc((Nv+1) * sizeof(double));
    //Y3[i] = (double*) malloc((Nv+1) * sizeof(double));
    G0nm1[i] = (double*) malloc((Nv+1) * sizeof(double));
    G1nm1[i] = (double*) malloc((Nv+1) * sizeof(double));
    G2nm1[i] = (double*) malloc((Nv+1) * sizeof(double));
    //G3nm1[i] = (double*) malloc((Nv+1) * sizeof(double));
    //G0n[i] = (double*) malloc((Nv+1) * sizeof(double));
    G1n[i] = (double*) malloc((Nv+1) * sizeof(double));
    G2n[i] = (double*) malloc((Nv+1) * sizeof(double));
    //G3n[i] = (double*) malloc((Nv+1) * sizeof(double));
}

// Memory allocation of matrix (=3-vectors).
MatrixA0nm1 = (double***) malloc((Ns+1) * sizeof(double**));
MatrixA1nm1 = (double***) malloc((Ns+1) * sizeof(double**));
MatrixA2nm1 = (double***) malloc((Ns+1) * sizeof(double**));
//MatrixA3nm1 = (double***) malloc((Ns+1) * sizeof(double**));
//MatrixA0n = (double***) malloc((Ns+1) * sizeof(double**));
MatrixA1n = (double***) malloc((Ns+1) * sizeof(double**));
MatrixA2n = (double***) malloc((Ns+1) * sizeof(double**));
//MatrixA3n = (double***) malloc((Ns+1) * sizeof(double**));
for (i=0; i<Ns+1; i++)
{
    MatrixA0nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
    MatrixA1nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
    MatrixA2nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
    //MatrixA3nm1[i] = (double**) malloc((Nv+1) * sizeof(double*));
    //MatrixA0n[i] = (double**) malloc((Nv+1) * sizeof(double*));
```

```
        MatrixA1n[i] = (double**) malloc((Nv+1) * sizeof(double*));
        MatrixA2n[i] = (double**) malloc((Nv+1) * sizeof(double*));
        //MatrixA3n[i] = (double**) malloc((Nv+1) * sizeof(double*));
        for (j=0; j<Nv+1; j++)
        {
            MatrixA0nm1[i][j] = (double*) malloc(11 * sizeof(double));
            MatrixA1nm1[i][j] = (double*) malloc(11 * sizeof(double));
            MatrixA2nm1[i][j] = (double*) malloc(11 * sizeof(double));
            //MatrixA3nm1[i][j] = (double*) malloc(11 * sizeof(double));
            //MatrixA0n[i][j] = (double*) malloc(11 * sizeof(double));
            MatrixA1n[i][j] = (double*) malloc(11 * sizeof(double));
            MatrixA2n[i][j] = (double*) malloc(11 * sizeof(double));
            //MatrixA3n[i][j] = (double*) malloc(11 * sizeof(double));
        }
    }
}// End of memory allocations.

// Initialization.
for (i=0; i<Ns+1; i++)
{
    for (j=0; j<Nv+1; j++)
    {
        // TERMINAL CONDITION FOR CALL AND PUT
        Unm1[i][j] = MAX(call_or_put*(sgrid[i] - strike),0.0);
        U_time[Nt][i][j] = Unm1[i][j];

        Utmp[i][j] = 0;
        Y0[i][j] = 0.;
        Y1[i][j] = 0.;
        //Y2[i][j] = 0.;
        //Y3[i][j] = 0.;
        G0nm1[i][j] = 0.;
        G1nm1[i][j] = 0.;
        G2nm1[i][j] = 0.;
        //G3nm1[i][j] = 0.;
        //G0n[i][j] = 0.;
        G1n[i][j] = 0.;
        G2n[i][j] = 0.;
        //G3n[i][j] = 0.;
    }
}
```

```
for (i=0; i<Ns+1; i++)
{
    for (j=0; j<Nv+1; j++)
    {
        for (st=0; st<11; st++)
        {
            MatrixA0nm1[i][j][st] = 0.;
            MatrixA1nm1[i][j][st] = 0.;
            MatrixA2nm1[i][j][st] = 0.;
            //MatrixA3nm1[i][j][st] = 0.;
            //MatrixA0n[i][j][st] = 0.;
            MatrixA1n[i][j][st] = 0.;
            MatrixA2n[i][j][st] = 0.;
            //MatrixA3n[i][j][st] = 0.;
        }

    }
}

// Temporal loop
for (time_index=Nt; time_index>0; time_index--)
{

    // Time Step
    deltat=tgrid[time_index]-tgrid[time_index-1];

    // American condition : Unknowns >= payoff
    if (am==1)
    {
        for (i=0; i<Ns+1; i++)
        {
            for (j=0; j<Nv+1; j++)
            {
                    if (call_or_put==1) //Call
                    {
                        Unm1[i][j] = MAX(Unm1[i][j],MAX(sgrid[i] - strike,0.
                    }
                    if (call_or_put==-1) //Put
                    {
                        Unm1[i][j] = MAX(Unm1[i][j],MAX(strike - sgrid[i],0.
```

```
                }
            }
        }
    }

    // Compute the elements at time step time_index-1, which is in the tau-f
    // (except A0n and G0n which are not used, so they are erased by the nex
    build_all_matrix(S0, sgrid, Ns,
                     V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                     R0, rho_sv,
                     call_or_put, strike, dividend,
                     tgrid, Nt, time_index-1,
                     MatrixA0nm1, MatrixA1n, MatrixA2n,
                     G0nm1, G1n, G2n);
    // Compute the elements at time step time_index.
    build_all_matrix(S0, sgrid, Ns,
                     V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
                     R0, rho_sv,
                     call_or_put, strike, dividend,
                     tgrid, Nt, time_index,
                     MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
                     G0nm1, G1nm1, G2nm1);


    if (scheme==0) // Douglas scheme
    {
        //Y0 = U[n-1] + Dt (A0+A1+A2+A3)[n-1] * U[n-1] + Dt (G0+G1+G2+G3)[n-
        compute_explicit_syslin_all_matrix(deltat,
                                           sgrid, Ns, vgrid, Nv,
                                           MatrixA0nm1, MatrixA1nm1, MatrixA
                                           G0nm1, G1nm1, G2nm1,
                                           Unm1, Y0);
        //Utmp = Y0 - theta*Dt A1[n-1] * U[n-1] - theta*Dt G1[n-1]
        computation_explicit_syslin_spot_matrix(theta * deltat,
                                                sgrid, Ns, vgrid, Nv,
                                                MatrixA0nm1, MatrixA1nm1, Ma
                                                G0nm1, G1nm1, G2nm1,
                                                Unm1, Y0, Utmp);
        //Y1 = (Id - theta*Dt A1[n])^-1 ( Utmp + theta*Dt G1[n] )
        computation_implicit_syslin_spot_matrix(theta * deltat,
                                                sgrid, Ns, vgrid, Nv,
```

```
                                                    MatrixA0nm1, MatrixA1n, Matr
                                                    G0nm1, G1n, G2n,
                                                    Utmp, Y1);
            //Utmp = Y1 - theta*Dt A2[n-1] * U[n-1] - theta*Dt G2[n-1]
            computation_explicit_syslin_var_matrix(theta * deltat,
                                                   sgrid, Ns, vgrid, Nv,
                                                   MatrixA0nm1, MatrixA1nm1, Mat
                                                   G0nm1, G1nm1, G2nm1,
                                                   Unm1, Y1, Utmp);
            //Y2 = (Id - theta*Dt A2[n])^-1 ( Utmp + theta*Dt G2[n] )
            computation_implicit_syslin_var_matrix(theta * deltat,
                                                   sgrid, Ns, vgrid, Nv,
                                                   MatrixA0nm1, MatrixA1n, Matri
                                                   G0nm1, G1n, G2n,
                                                   Utmp, Unm1);// /!\  with Y2=U
        }

        if (scheme==1) // Craig-Sneyd scheme
        {

        }
        if (scheme==2) // Modified Craig-Sneyd scheme
        {

        }
        if (scheme==3) // Hundsdorfer-Verwer scheme
        {

        }

        // End of time step. Copy values in Utmp and in U_time
        for (i=0; i<Ns+1; i++)
        {
            for (j=0; j<Nv+1; j++)
            {
                Utmp[i][j] =  MAX(Unm1[i][j],0.);
                U_time[time_index-1][i][j] = Utmp[i][j];
            }
        }
    }
```

```
// Memory desallocations.
{
    // Memory desallocation of matrix (=4-vectors).
    for (i=0; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            free(MatrixA0nm1[i][j]);
            free(MatrixA1nm1[i][j]);
            free(MatrixA2nm1[i][j]);
            //free(MatrixA3nm1[i][j]);
            //free(MatrixA0n[i][j]);
            free(MatrixA1n[i][j]);
            free(MatrixA2n[i][j]);
            //free(MatrixA3n[i][j]);
        }
        free(MatrixA0nm1[i]);
        free(MatrixA1nm1[i]);
        free(MatrixA2nm1[i]);
        //free(MatrixA3nm1[i]);
        //free(MatrixA0n[i]);
        free(MatrixA1n[i]);
        free(MatrixA2n[i]);
        //free(MatrixA3n[i]);
    }
    free(MatrixA0nm1);
    free(MatrixA1nm1);
    free(MatrixA2nm1);
    //free(MatrixA3nm1);
    //free(MatrixA0n);
    free(MatrixA1n);
    free(MatrixA2n);
    //free(MatrixA3n);

    // Memory desallocation of 3-vectors.
    for (i=0; i<Ns+1; i++)
    {
        free(Unm1[i]);
        free(Utmp[i]);
        free(Y0[i]);
        free(Y1[i]);
```

```
            //free(Y2[i]);
            //free(Y3[i])
            free(G0nm1[i]);
            free(G1nm1[i]);
            free(G2nm1[i]);
            //free(G3nm1[i]);
            //free(G0n[i]);
            free(G1n[i]);
            free(G2n[i]);
            //free(G3n[i]);
        }

        free(Unm1);
        free(Utmp);
        free(Y0);
        free(Y1);
        //free(Y2);
        //free(Y3)
        free(G0nm1);
        free(G1nm1);
        free(G2nm1);
        //free(G3nm1);
        //free(G0n);
        free(G1n);
        free(G2n);
        //free(G3n);
    }// End of memory desallocations.
}

static void make_interpolation_in_all_prices(double St, double *sgrid, int Ns,
                                             double Vt, double *vgrid, int Nv,
                                             int time_index, double*** U_time,
                                             double *ptprice)
{
    int indexS, indexV;
    double price_sd_vd, price_sd_vu;
    double price_su_vd, price_su_vu;

    // Find the index in sgrid corresponding to the price St of the asset.
    indexS = lower_index(sgrid,Ns+1,St);
    // Find the index in vgrid corresponding to the variance Vt of the asset.
```

```
    indexV = lower_index(vgrid,Nv+1,Vt);

    // Compute the price. We do an interpolation.
    price_sd_vd = U_time[time_index][indexS][indexV];
    price_sd_vu = U_time[time_index][indexS][indexV+1];
    price_su_vd = U_time[time_index][indexS+1][indexV];
    price_su_vu = U_time[time_index][indexS+1][indexV+1];
    *ptprice = double_interpolation(price_sd_vd, price_sd_vu, price_su_vd, price
                                    sgrid[indexS], sgrid[indexS+1],
                                    vgrid[indexV], vgrid[indexV+1],
                                    St, Vt);
}

////////////////////////////////////////////////////////////////////////////////
//Compute CVA.
////////////////////////////////////////////////////////////////////////////////
static void compute_CVA(double S0, double *sgrid, int Ns,
                        double V0, double sigma_v, double alpha_v, double beta_v
                        double *vgrid, int Nv,
                        double R0, double rho_sv,
                        int call_or_put, double strike, double dividend, int am,
                        double recovery_rate, double default_intensity,
                        double *tgrid, int Nt,
                        long NMC, double confidence, int flag_cir,
                        double theta,  int scheme,int generator,
                        double *ptprice)



{
    // Variables for loops.
    int i, j, time_index;
    long ipath;

    // Variables to compute price and delta.
    double St, Vt;
    double ***U_time;
    double *price_time;
    double factor;

    // Variables for Monte-Carlo procedure
```

```
double alpha, z_alpha;
double g1,g2;
double h;
double sqrt_h;
double *X1a,*X2a,*X3a,*X4a;
double w_t_1,w_t_2;
double aaa;
double Kseuil,aux;
double mu;

h= tgrid[Nt]-tgrid[Nt-1]; // Time step
sqrt_h = sqrt(h);
mu=R0-dividend;
aaa=alpha_v*beta_v;

// Memory allocation of timed arrays
price_time = (double*) malloc((Nt+1) * sizeof(double));

U_time = (double***) malloc((Nt+1) * sizeof(double**));
for (time_index=0; time_index<Nt+1; time_index++)
{
    U_time[time_index] = (double**) malloc((Ns+1) * sizeof(double*));
    for (i=0; i<Ns+1; i++)
    {
        U_time[time_index][i] = (double*) malloc((Nv+1) * sizeof(double));
    }
}

// Initialization
for (time_index=0; time_index<Nt+1; time_index++)
{
    price_time[time_index]=0.;

    for (i=0; i<Ns+1; i++)
    {
        for (j=0; j<Nv+1; j++)
        {
            U_time[time_index][i][j] = 0.;
        }
    }
}
```

```
// Compute all prices
compute_all_prices(S0, sgrid, Ns,
                   V0,  sigma_v, alpha_v, beta_v,
                   vgrid, Nv,
                   R0, rho_sv,
                   call_or_put, strike, dividend, am,
                   tgrid, Nt,
                   theta,  scheme,
                   U_time);


// Random parameters
if(flag_cir==1)
    Kseuil=MAX((0.25*SQR(sigma_v)-aaa)*psik(h*0.5,alpha_v),0.);
else
{
    if (alpha_v==0)
        Kseuil=1;
    else Kseuil=(exp(alpha_v*h)-1)/(h*alpha_v);
    if (sigma_v*sigma_v <= 4*alpha_v*beta_v/3) {

        Kseuil=Kseuil*sigma_v*sqrt(alpha_v*beta_v-sigma_v*sigma_v/4)/sqrt(2)
    }
    if (sigma_v*sigma_v > 4*alpha_v*beta_v/3 && sigma_v*sigma_v <= 4*alpha_v
        aux=(0.5*sigma_v*sqrt(3+sqrt(6))+sqrt(sigma_v*sigma_v/4 -
                                              alpha_v*beta_v+sigma_v*sqrt(-s
        Kseuil=Kseuil*SQR(aux);
    }
    if (sigma_v*sigma_v > 4*alpha_v*beta_v){
        aux=0.5*sigma_v*sqrt(3+sqrt(6))+ sqrt(sigma_v*sqrt(sigma_v*sigma_v/4
        Kseuil=Kseuil*(sigma_v*sigma_v/4 - alpha_v*beta_v + SQR(aux));
    }
    if (sigma_v*sigma_v == 4*alpha_v*beta_v) Kseuil=0;
}

// Value to construct the confidence interval
alpha= (1.- confidence)/2.;
z_alpha= pnl_inv_cdfnor(1.- alpha);
```

```
// Memory allocation
X1a = (double*)malloc(sizeof(double)*(Nt+1));
X2a = (double*)malloc(sizeof(double)*(Nt+1));
X3a = (double*)malloc(sizeof(double)*(Nt+1));
X4a = (double*)malloc(sizeof(double)*(Nt+1));

// Monte-Carlo loop
for(ipath= 1;ipath<= NMC;ipath++)
{
    // Initialization at time 0
    X1a[0]=V0; X2a[0]=0; X3a[0]=S0;  X4a[0]=0;
    // Find price at value (St,Vt)
    Vt = X1a[0];
    St = X3a[0];
    make_interpolation_in_all_prices(St, sgrid, Ns,
                                     Vt, vgrid, Nv,
                                     0, U_time,
                                     ptprice);
    // Sum the values
    price_time[0] += *ptprice;

    // Time loop
    for(time_index=1 ; time_index<=Nt ; time_index++)
    {
        // Discrete law obtained by matching of first five moments of a gaus
        if(flag_cir==1)
            g1=DiscLawMatch5(generator);
        else
            g1=DiscLawMatch7(generator);
        w_t_1=sqrt_h*g1;
        g2= pnl_rand_normal(generator);
        w_t_2=sqrt_h*g2;
        X1a[time_index]=X1a[time_index-1];
        X2a[time_index]=X2a[time_index-1];
        X3a[time_index]=X3a[time_index-1];
        X4a[time_index]=X4a[time_index-1];
        fct_Heston(&X1a[time_index],&X2a[time_index],&X3a[time_index],&X4a[t
                   h,w_t_1,w_t_2,aaa,alpha_v,sigma_v,mu,rho_sv,Kseuil,genera

        // Find price at value (St,Vt)
        Vt = X1a[time_index];
```

```
        St = X3a[time_index];
        make_interpolation_in_all_prices(St, sgrid, Ns,
                                         Vt, vgrid, Nv,
                                         time_index, U_time,
                                         ptprice);
        // Sum the values
        price_time[time_index] += *ptprice;
    }
} // End of Monte-Carlo loop

// Compute estimators
for(time_index=1 ; time_index<=Nt ; time_index++)
{
    // Price estimator
    price_time[time_index] = (price_time[time_index]/((double)NMC));
    price_time[time_index] = exp(-R0*tgrid[time_index]) * price_time[time_in

}

// Compute time integral
*ptprice=0.;
for(time_index=1 ; time_index<=Nt ; time_index++)
{
    factor = (1.-recovery_rate) *
    (exp(-default_intensity * tgrid[time_index-1]) - exp(-default_intensity*

    *ptprice += price_time[time_index] * factor;
}

// Memory desallocation
free(X1a);
free(X2a);
free(X3a);
free(X4a);
free(price_time);


for (time_index=0; time_index<Nt+1; time_index++)
{
    for (i=0; i<Ns+1; i++)
    {
```

```
            free(U_time[time_index][i]);
        }
        free(U_time[time_index]);
    }
    free(U_time);

}


int FDAdi_CVA(int am,double S0, NumFunc_1  *p, double maturity, double strike, d
{
  int call_or_put,i;
  double ptprice;
  double *sgrid, *vgrid, *tgrid;
    double Smax,Vmax,Sleft,Sright,Coeff_s,Coeff_v;
    int scheme;
    double theta;
    double confidence;
    int flag_cir;
     PnlRng *rng;
   confidence=0.95;
   flag_cir=1;

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

  rng = pnl_rng_create(generator);
  pnl_rng_sseed(rng, 0);


  // Spot
    Smax = 5.*S0;
    Sleft = 0.8*S0;
    Sright = 1.2*S0;
    Coeff_s = S0/20; // Environ entre 2 et 5 ou fraction de Ns/2
    // Variance
    Vmax = -MAX(-MAX(Nv*V0,1.),-5.);
    Coeff_v = Vmax/500;
    // Scheme numerical parameters.
```

```
    theta=0.5;//Theta schema
    scheme=0;//Douglas Scheme

    ////////////////////////////////////
    // Compute sgrid, vgrid and tgrid. //
    ////////////////////////////////////
    // Memory allocation of 1-vectors.
    sgrid=(double *)malloc((Ns+1)*sizeof(double));
    vgrid=(double *)malloc((Nv+1)*sizeof(double));
    tgrid=(double *)malloc((Nt+1)*sizeof(double));

    grid_generation_spot(sgrid, Sleft, Sright, Smax, Ns, Coeff_s);
    grid_generation_variance(vgrid, Vmax, Nv, Coeff_v, V0);
    for (i=0; i<=Nt; i++) tgrid[i] = i*maturity/((double)Nt);

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

    compute_CVA(S0, sgrid, Ns,
                V0, sigma_v, alpha_v, beta_v,
                vgrid, Nv,
                R0, rho_sv,
                call_or_put, strike, dividend, am,
                recovery_rate, default_intensity,
                tgrid, Nt,
                NMC, confidence, flag_cir,
                theta, scheme,generator,
                &ptprice);


    free(sgrid);
    free(vgrid);
    free(tgrid);

//CVA
  *CVA=ptprice;

  return OK;

}
```

```
int CALC(FD_Adi_HestonCVA)(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 FDAdi_CVA(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_Adi_HestonCVA)(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_Adi1_HestonCVA";
  Met->Par[0].Val.V_PINT = 40;
```

```
  Met->Par[1].Val.V_PINT = 100;
      Met->Par[2].Val.V_PINT = 20;
      Met->Par[3].Val.V_LONG = 15000;
      Met->Par[4].Val.V_ENUM.value = 0;
      Met->Par[4].Val.V_ENUM.members = &PremiaEnumMCRNGs;
    }


  return OK;
}


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