Help

```
#include  "
href../../../mod/merhes1d/merhes1d_std/merhes1d_std_h_src.pdfmerhes1d_std.h"
#include "
href../../../common/enums_h_src.pdfenums.h"

#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_tridiag_matrix.h"
#include "pnl/pnl_band_matrix.h"

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


//////////
// 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]);
    }

  //    if (V0_IN_GRID)
```

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

static void grid_generation_pide(double *pidegrid, double tau, int Npide)
{
  int k;
  double deltat;

  deltat = tau/(double)Npide;
  // Definition of uniform grid.
  for (k=0; k<=Npide; k++)
    {
  pidegrid[k] = k * deltat;
    }
}


//////////
// 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) + valu
}

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 ste
    {
  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)
{
  return call_or_put * 0.;
```

```
}

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)
{
  //    return (NEUMANN_ONE) ? 0. : S0*exp(sgrid[Ns]);
  return call_or_put * 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, double jump_convection,
                             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=0;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;

  // PDE in logarithm formulation
  convection_s = actualrpoint-dividend-actualvpoint/2.0-jump_convection;
  diffusion_s = actualvpoint/2.0;
  convection_v = alpha_v*(beta_v - actualvpoint); //convection_v = kappa*(eta -
  diffusion_v = sigma_v * sigma_v * actualvpoint/2.0;
  order_0 = -actualrpoint;
  mixted_sv = rho_sv * sigma_v * actualvpoint;

  ////////////////////
  // Diffusion and Convection S
  ////////////////////
  {
if (i==0) // S=Smin -> Neumann
  {
Dsip1 = sgrid[i+1]-sgrid[i];
Dsi = Dsip1;
//ds[stencil(-1,0)]=2.0/(Dsi*(Dsi+Dsip1));
ds[stencil(0,0)]=-2.0/(Dsi*Dsip1) + 2.0/(Dsi*(Dsi+Dsip1));
ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1));
G_ds += 2.0/(Dsi*(Dsi+Dsip1)) * bc_spot_min(S0, sgrid, Ns, i,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);

//cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1));
cs[stencil(0,0)]=(Dsip1-Dsi)/(Dsi*Dsip1) - Dsip1/(Dsi*(Dsi+Dsip1));
cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1));
G_cs += -Dsip1/(Dsi*(Dsi+Dsip1)) * bc_spot_min(S0, sgrid, Ns, i,
```

```
   V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
   R0, rho_sv,
   call_or_put, strike, dividend,
   tgrid, Nt, time_index);
  }
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+Dsip1));
//ds[stencil(1,0)]=2.0/(Dsip1*(Dsi+Dsip1));
G_ds = 2.0/(Dsip1*(Dsi+Dsip1)) * bc_spot_max(S0, sgrid, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);

cs[stencil(-1,0)]=-Dsip1/(Dsi*(Dsi+Dsip1));
cs[stencil(0,0)]=(Dsip1-Dsi)/(Dsi*Dsip1) + Dsi/(Dsip1*(Dsi+Dsip1));
//cs[stencil(1,0)]=Dsi/(Dsip1*(Dsi+Dsip1));
G_cs += Dsi/(Dsip1*(Dsi+Dsip1)) * bc_spot_max(S0, sgrid, Ns, i,
  V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
  R0, rho_sv,
  call_or_put, strike, dividend,
  tgrid, Nt, time_index);
  }
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+Dvjp1));
//dv[stencil(0,1)]=2.0/(Dvjp1*(Dvj+Dvjp1));
G_dv += 2.0/(Dvjp1*(Dvj+Dvjp1)) * bc_var_max(S0, sgrid, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);

// 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 sign.
if ((convection_v<0) && (j>1)) // var > beta_v and j>1 -> Backward.
  {
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
  {
// If S=0 or V=0 ->> misted_sv = 0.

// 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 of V.
// 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 of S.
```

```
if ((0<i) && (i<Ns) && (0<j) && (j<Nv)) // Strict interior 0<i<Ns and 0<j<Nv
  {
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==0 or i==Ns or j==0 or j==Nv
  {
if (j==Nv) // V=Vmax
  {
// Three cases :
// i==0 -> Condition on Vmin
if (i==0) // S=Smin -> Neumann
  {
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, sgrid, Ns, i,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);
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, Ns, i,
   V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
   R0, rho_sv,
   call_or_put, strike, dividend,
   tgrid, Nt, time_index);
//msv[stencil(-1,-1)] = sbeta_im1 * vbeta_jm1;
G_msv += sbeta_im1 * vbeta_jm1 * bc_spot_min(S0, sgrid, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j-1,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);
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, sgrid, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j+1,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);
//msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1;
G_msv += sbeta_ip1 * vbeta_jp1 * bc_var_max(S0, sgrid, Ns, i+1,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);

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

```
  + 1. * sbeta_im1 * vbeta_jm1 // Neumann
  + 1. * sbeta_im1 * vbeta_jp1 // Neumann
  + 1. * sbeta_ip1 * vbeta_jp1; // Neumann
  }
// 0<i<Ns -> Condition on V=Vmax.
// i==Ns -> Condition on V=Vmax compatible with condition on S=Smax.
if ((0<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, Ns, i,
   V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
   R0, rho_sv,
   call_or_put, strike, dividend,
   tgrid, Nt, time_index);
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, sgrid, Ns, i-1,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);
//msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1;
G_msv += sbeta_ip1 * vbeta_jp1 * bc_var_max(S0, sgrid, Ns, i+1,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
```

```
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);
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, sgrid, Ns, i,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);

//msv[stencil(0,-1)] = sbeta_i0 * vbeta_jm1;
G_msv += sbeta_i0 * vbeta_jm1 * bc_spot_max(S0, sgrid, Ns, i,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j-1,
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);
```

```
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, sgrid, Ns, i-1,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);
//msv[stencil(1,-1)] = sbeta_ip1 * vbeta_jm1;
G_msv += sbeta_ip1 * vbeta_jm1 * bc_spot_max(S0, sgrid, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);
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, sgrid, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j+1,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);
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 condition on V=Vmax.
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, sgrid, Ns, i,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);
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, sgrid, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j-1,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);
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, sgrid, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j+1,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);
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 it now !
if ((i==Ns) && (j==Nv)) // Corner ! The two Neumann conditions are compatible.
  {
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, Ns, i,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);
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, i,
   V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
   R0, rho_sv,
   call_or_put, strike, dividend,
   tgrid, Nt, time_index);
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, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j-1,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);
//msv[stencil(-1,1)] = sbeta_im1 * vbeta_jp1;
G_msv += sbeta_im1 * vbeta_jp1 * bc_var_max(S0, sgrid, Ns, i,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j+1,
R0, rho_sv,
call_or_put, strike, dividend,
tgrid, Nt, time_index);
//msv[stencil(1,1)] = sbeta_ip1 * vbeta_jp1; !!!! Here COMPATIBILITY is OK !!!!
G_msv += sbeta_ip1 * vbeta_jp1 * bc_spot_max(S0, sgrid, Ns, i,
 V0, sigma_v, alpha_v, beta_v, vgrid, Nv, j,
 R0, rho_sv,
 call_or_put, strike, dividend,
 tgrid, Nt, time_index);
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_s * ds[0]
  MatrixA2[i][j][0] = order_0 * 0.5 + convection_v * cv[0] + diffusion_v * dv[0]

  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=0; 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=0; 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=0 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, 1.0, 0.0); // I
  Working_Matrix = pnl_tridiag_mat_create(Ns+1);
  Entree = pnl_vect_create(Ns+1);
  Sortie = pnl_vect_create(Ns+1);

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

```
  {
// Build the matrix

//pnl_tridiag_mat_set (Working_Matrix, 0, -1, MatrixA1[0][j][1]);
pnl_tridiag_mat_set (Working_Matrix, 0,  0, MatrixA1[0][j][0]);
pnl_tridiag_mat_set (Working_Matrix, 0,  1, MatrixA1[0][j][2]);
for (i=0; i<Ns-1; i++)
      {
pnl_tridiag_mat_set (Working_Matrix, i+1, -1, MatrixA1[i+1][j][1]);
pnl_tridiag_mat_set (Working_Matrix, i+1,  0, MatrixA1[i+1][j][0]);
pnl_tridiag_mat_set (Working_Matrix, i+1,  1, MatrixA1[i+1][j][2]);
      }
pnl_tridiag_mat_set (Working_Matrix, Ns, -1, MatrixA1[Ns][j][1]);
pnl_tridiag_mat_set (Working_Matrix, Ns,  0, MatrixA1[Ns][j][0]);
//pnl_tridiag_mat_set (Working_Matrix, Ns,  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+1; i++)
      {
pnl_vect_set (Entree, i, rhs[i][j] + coeff * G1[i][j]);
      }

pnl_tridiag_mat_syslin(Sortie, Working_Matrix, Entree);

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

  // 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);
}


static void compute_jump_equation(double *sgrid, int Ns, double *pidegrid, int N
                                  double mu_J, double sigma_J, double lambda_J,
                                  double bound_min, double bound_max,
                                  double *initial_data, double *final_data)
{

  int i,j;
  double Dsi;
  double Dsip1;
  double delta_time;
  double diff;
```

```
double* tmp_j_data;
double* tmp_jp1_data;
double Neumann;
double bound_min_local;
double bound_max_local;
double *AL, *BL, *CL, *AR, *BR, *CR;

tmp_j_data = (double*)malloc((Ns+1)*sizeof(double));
tmp_jp1_data = (double*)malloc((Ns+1)*sizeof(double));

diff = 0.5 * sigma_J*sigma_J;
// Solve the pde d_s Z = L z on s in [0,1]
// Z_n+1 = Z_n + delta_s L Z_n
for (i=0; i<Ns+1; i++)
  {
tmp_j_data[i] = initial_data[i];
  }
Neumann = (tmp_j_data[Ns]-tmp_j_data[Ns-1])/(sgrid[Ns]-sgrid[Ns-1]);

AL= (double*)malloc((Ns+1)*sizeof(double));
BL= (double*)malloc((Ns+1)*sizeof(double));
CL= (double*)malloc((Ns+1)*sizeof(double));
AR= (double*)malloc((Ns+1)*sizeof(double));
BR= (double*)malloc((Ns+1)*sizeof(double));
CR= (double*)malloc((Ns+1)*sizeof(double));


// Time loop
for (j=0; j<Npide; j++)
  {
Neumann = (tmp_j_data[Ns]-tmp_j_data[Ns-1])/(sgrid[Ns]-sgrid[Ns-1]);

// Fill the left and right matrix
delta_time = pidegrid[j+1]-pidegrid[j];
for (i=0; i<Ns+1; i++)
      {
if (i==0)
          {
Dsip1 = sgrid[i+1]-sgrid[i];
Dsi = Dsip1;
          }
```

```
else if (i==Ns)
          {
Dsi = sgrid[i]-sgrid[i-1];
Dsip1 = Dsi;
          }
else
          {
Dsip1 = sgrid[i+1]-sgrid[i];
Dsi = sgrid[i]-sgrid[i-1];
          }

// Peclet Condition-Coefficient of diffusion augmented
if ((Dsi*fabs(mu_J))<=2*diff)
          {
// Centered
AL[i]=-0.5 * delta_time * (2.0/(Dsi*(Dsi+Dsip1)) * diff -Dsip1/(Dsi*(Dsi+Dsip1
BL[i]=1. - 0.5 * delta_time * (-2.0/(Dsi*Dsip1) * diff + (Dsip1-Dsi)/(Dsi*Dsip
CL[i]=-0.5 * delta_time * (2.0/(Dsip1*(Dsi+Dsip1)) * diff + Dsi/(Dsip1*(Dsi+Ds
AR[i]=0.5 * delta_time * (2.0/(Dsi*(Dsi+Dsip1)) * diff -Dsip1/(Dsi*(Dsi+Dsip1)
BR[i]=1. + 0.5 * delta_time * (-2.0/(Dsi*Dsip1) * diff + (Dsip1-Dsi)/(Dsi*Dsip
CR[i]=0.5 * delta_time * (2.0/(Dsip1*(Dsi+Dsip1)) * diff + Dsi/(Dsip1*(Dsi+Dsi
          }
else
          {
if (mu_J>0.) // Upwind
               {
AL[i]=-0.5 * delta_time * (2.0/(Dsi*(Dsi+Dsip1)) * diff );
BL[i]=1. - 0.5 * delta_time * (-2.0/(Dsi*Dsip1) * diff - 1./Dsip1 * mu_J);
CL[i]=-0.5 * delta_time * (2.0/(Dsip1*(Dsi+Dsip1)) * diff + 1./Dsip1 * mu_J);
AR[i]=0.5 * delta_time * (2.0/(Dsi*(Dsi+Dsip1)) * diff );
BR[i]=1. + 0.5 * delta_time * (-2.0/(Dsi*Dsip1) * diff - 1./Dsip1 * mu_J);
CR[i]=0.5 * delta_time * (2.0/(Dsip1*(Dsi+Dsip1)) * diff + 1./Dsip1 * mu_J);
               }
else // Downwind
               {
AL[i]=-0.5 * delta_time * (2.0/(Dsi*(Dsi+Dsip1)) * diff - 1./Dsi * mu_J);
BL[i]=1. - 0.5 * delta_time * (-2.0/(Dsi*Dsip1) * diff + 1./Dsi * mu_J);
CL[i]=-0.5 * delta_time * (2.0/(Dsip1*(Dsi+Dsip1)) * diff );
AR[i]=0.5 * delta_time * (2.0/(Dsi*(Dsi+Dsip1)) * diff - 1./Dsi * mu_J);
BR[i]=1. + 0.5 * delta_time * (-2.0/(Dsi*Dsip1) * diff + 1./Dsi * mu_J);
CR[i]=0.5 * delta_time * (2.0/(Dsip1*(Dsi+Dsip1)) * diff );
```

```
                }

            }
        }
  // Make index i=0 and i=Ns
  BR[0]=BR[0]+AR[0];
  BL[0]=BL[0]+AL[0];
  BR[Ns]=BR[Ns]+CR[Ns];
  BL[Ns]=BL[Ns]+CL[Ns];

  // Set the Gauss algorithm
  for(i=Ns-1;i>=0;i--)
BL[i]=BL[i]-CL[i]*AL[i+1]/BL[i+1];
  for(i=0;i<Ns-1;i++)
AL[i]=AL[i]/BL[i];
  for(i=0;i<Ns;i++)
CL[i]=CL[i]/BL[i+1];

  // FD Cycle
  bound_min_local = bound_min * (sgrid[1]-sgrid[0]);
  bound_max_local = bound_max * (sgrid[Ns]-sgrid[Ns-1]);

  tmp_jp1_data[0]=BR[0]*tmp_j_data[0]+CR[0]*tmp_j_data[1]+AR[0]*bound_min_local-
  for(i=1;i<Ns;i++)
tmp_jp1_data[i]=AR[i]*tmp_j_data[i-1]+
  BR[i]*tmp_j_data[i]+
  CR[i]*tmp_j_data[i+1];
  tmp_jp1_data[Ns]=AR[Ns]*tmp_j_data[Ns-1]+BR[Ns]*tmp_j_data[Ns]+CR[Ns]*bound_ma

  // Solve the system
  for(i=Ns-1;i>=0;i--)
tmp_jp1_data[i]=tmp_jp1_data[i]-CL[i]*tmp_jp1_data[i+1];

  tmp_j_data[0] =tmp_jp1_data[0]/BL[0];
  for(i=1;i<Ns+1;i++)
tmp_j_data[i]=tmp_jp1_data[i]/BL[i]-AL[i]*tmp_j_data[i-1];


  // Neumann condition
  //tmp_j_data[0] = tmp_j_data[1]; // Homogeneous
  //tmp_j_data[Ns] = tmp_j_data[Ns-1] + (sgrid[Ns]-sgrid[Ns-1]) *  Neumann;
```

```
    }

  // Add the initial data to obtain J C = lambda (-1 + exp(L)) C
  for (i=0; i<Ns+1; i++)
    {
  final_data[i] = -lambda_J * initial_data[i] + lambda_J * tmp_j_data[i];
    }

  free(tmp_j_data);
  free(tmp_jp1_data);
  free(AL);
  free(BL);
  free(CL);
  free(AR);
  free(BR);
  free(CR);
}

static void solve_jump_equation(double *sgrid, int Ns, double coeff,
                                double * pidegrid, int Npide,
                                double mu_J, double sigma_J, double lambda_J, do
                                double bound_min, double bound_max,
                                double *initial_data, double *final_data)
{
  int picard_index;
  int i;
  double* rhs_data;
  double* tmp_data;
  double error;

  // Compute RHS
  tmp_data = (double*)malloc((Ns+1)*sizeof(double));
  rhs_data = (double*)malloc((Ns+1)*sizeof(double));
  compute_jump_equation(sgrid, Ns, pidegrid, Npide,
mu_J, sigma_J, lambda_J, kappa_J,
bound_min, bound_max,
initial_data, tmp_data);
  for (i=0; i<Ns+1; i++)
    {
  rhs_data[i] = initial_data[i] + 0.5 * coeff * tmp_data[i];
```

```
    }

  // Initialize Picard iterations
  for (i=0; i<Ns+1; i++)
    {
  final_data[i] = initial_data[i];
    }
  picard_index=0;
  error=100;
  // Make Picard iterations
  while ((++picard_index<10) & (error>10.))
    {
  // Save old values
  for (i=0; i<Ns+1; i++)
        {
  initial_data[i] = final_data[i];
        }
  compute_jump_equation(sgrid, Ns, pidegrid, Npide,
mu_J, sigma_J, lambda_J, kappa_J,
bound_min, bound_max,
final_data, tmp_data);
  // Compute new values
  for (i=0; i<Ns+1; i++)
        {
  final_data[i] = rhs_data[i] + 0.5 * coeff * tmp_data[i];
        }
  // Compute error
  error = 0.;
  for (i=0; i<Ns+1; i++)
        {
  tmp_data[i] = fabs(final_data[i] - initial_data[i]);
  error = error + tmp_data[i];
        }
    }
  free(tmp_data);
  free(rhs_data);
}


static void compute_jumps_inline(double coeff,
                                 double *sgrid, int Ns, double *vgrid, int Nv,
```

```
                                    double *pidegrid, int Npide,
                                    double mu_J, double sigma_J, double lambda_J, d
                                    double bound_min, double bound_max,
                                    double **tab_2D)
{
  int i,j;
  double *initial_tab_1D;
  double *final_tab_1D;
  initial_tab_1D = (double *)malloc((Ns+1)*sizeof(double));
  final_tab_1D = (double *)malloc((Ns+1)*sizeof(double));

  // In all v-directions
  for (j=0; j<Nv+1; j++)
    {
  for (i=0; i<Ns+1; i++)
        {
  initial_tab_1D[i] = tab_2D[i][j];
        }
  // Solve the jump equation
  solve_jump_equation(sgrid, Ns, coeff, pidegrid, Npide,
  mu_J, sigma_J, lambda_J, kappa_J,
  bound_min, bound_max,
  initial_tab_1D, final_tab_1D);
  for (i=0; i<Ns+1; i++)
        {
  tab_2D[i][j] = final_tab_1D[i];
        }
    }
  free(initial_tab_1D);
  free(final_tab_1D);
}

////////////////////////////////////////////////////////////////////////////////
// Function to manage the time evolution
////////////////////////////////////////////////////////////////////////////////
static void time_evolution
(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,
 double *pidegrid, int Npide,
```

```
 double mu_J, double sigma_J, double lambda_J, double kappa_J,
 double *tgrid, int Nt,
 double theta, int scheme,
 // 2-vectors
 double **Unm1, double **Utmp, double **Y0, double **Y1, //double **Y2, //double
 double **G0nm1, double **G1nm1, double **G2nm1, //double **G3nm1,
 //double **G0n;
 double **G1n, double **G2n, //double **G3n,
 double **GSecondMember,
 // Matrix (=3-vectors)
 double ***MatrixA0nm1, double ***MatrixA1nm1, double ***MatrixA2nm1, //double *
 //double ***MatrixA0n;
 double ***MatrixA1n, double ***MatrixA2n) //double ***MatrixA3n)
{
  int time_index, i, j;
  //int st;
  double deltat;

  // Initialization
  for (i=0; i<Ns+1; i++)
    {
  for (j=0; j<Nv+1; j++)
        {
  Utmp[i][j] = 0;
  Y0[i][j] = 0.;
  Y1[i][j] = 0.;
  //Y2[i][j] = 0.;
  //Y3[i][j] = 0.;
        }
    }

  // Finite difference cycle in t-backward order and tau-forward order.
  for (time_index=Nt;time_index>0;time_index--)
    {
  // Time Step
  deltat=tgrid[time_index]-tgrid[time_index-1];


  // Build second member values
  for(i=0;i<Ns+1;i++)
{
```

```
  for(j=0;j<Nv+1;j++)
{
  G1n[i][j] = GSecondMember[i][j];
  G1nm1[i][j] = GSecondMember[i][j];
}
}


  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-1]
  compute_explicit_syslin_all_matrix(deltat * 0.5,
 sgrid, Ns, vgrid, Nv,
 MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
 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 * 0.5,
  sgrid, Ns, vgrid, Nv,
  MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
  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 * 0.5,
  sgrid, Ns, vgrid, Nv,
  MatrixA0nm1, MatrixA1n, MatrixA2n,
  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 * 0.5,
 sgrid, Ns, vgrid, Nv,
 MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
 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 * 0.5,
 sgrid, Ns, vgrid, Nv,
 MatrixA0nm1, MatrixA1n, MatrixA2n,
 G0nm1, G1n, G2n,
 Utmp, Unm1);
```

```
 // Compute jumps
 if (call_or_put==1)
           { // Call option
 compute_jumps_inline(deltat, sgrid, Ns, vgrid, Nv,
  pidegrid, Npide, mu_J, sigma_J, lambda_J, kappa_J,
  0., S0*exp(sgrid[Ns]),
  Unm1);
           }
 else
           { // Put option
 compute_jumps_inline(deltat, sgrid, Ns, vgrid, Nv,
  pidegrid, Npide, mu_J, sigma_J, lambda_J, kappa_J,
  strike*exp(-R0 * tgrid[Nt-time_index+1]),0.,
  Unm1);

           }
 //Y0 = U[n-1] + Dt (A0+A1+A2+A3)[n-1] * U[n-1] + Dt (G0+G1+G2+G3)[n-1]
 compute_explicit_syslin_all_matrix(deltat * 0.5,
sgrid, Ns, vgrid, Nv,
MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
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 * 0.5,
 sgrid, Ns, vgrid, Nv,
 MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
 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 * 0.5,
 sgrid, Ns, vgrid, Nv,
 MatrixA0nm1, MatrixA1n, MatrixA2n,
 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 * 0.5,
sgrid, Ns, vgrid, Nv,
MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
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 * 0.5,
sgrid, Ns, vgrid, Nv,
MatrixA0nm1, MatrixA1n, MatrixA2n,
G0nm1, G1n, G2n,
Utmp, Unm1); // /!\  with Y2=Un which becomes Unm1 for the next loop.

       }


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

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

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

       }
    }

  // End of temporal loop. Copy values in Utmp.
  for (i=0; i<Ns+1; i++)
    {
  for (j=0; j<Nv+1; j++)
       {
  Utmp[i][j] =  MAX(Unm1[i][j],0.);
       }
    }

}


////////////////////////////////////////////////////////////////////////////
//Compute option price and delta.
////////////////////////////////////////////////////////////////////////////
static void compute_price_and_delta(double S0, double *sgrid, int Ns,
                                    double V0, double sigma_v, double alpha_v, d
                                    double *vgrid, int Nv,
```

```
                                        double R0, double rho_sv,
                                        int call_or_put, double strike, double divid
                                        double *pidegrid, int Npide,
                                        double mu_J, double sigma_J, double lambda_J
                                        double *tgrid, int Nt,
                                        double theta,  int scheme,
                                        double *ptprice, double *ptdelta)
{
  // Variables for loops.
  int i, j, st;

  // Variables to compute price and delta.
  int IndexS, IndexV;
  double price_sd_vd, price_sd_vu;
  double price_su_vd, price_su_vu;

  // 2-vectors
  double **U_new;
  double **U_old;
  double **Y0;
  double **Y1;
  //double **Y2;
  //double **Y3;
  double **G0nm1;
  double **G1nm1;
  double **G2nm1;
  //double **G3nm1;
  //double **G0n;
  double **G1n;
  double **G2n;
  //double **G3n;
  double **GSecondMember;

  // 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.
U_old = (double**) malloc((Ns+1) * sizeof(double*));
U_new = (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++)
  {
U_old[i] = (double*) malloc((Nv+1) * sizeof(double));
U_new[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++)
        {
  U_old[i][j] = 0;
  U_new[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.;
            }

        }
    }




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

  // Construction before loop. Using last time... (Nt).
  build_all_matrix(S0, sgrid, Ns,
```

```
   V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
   R0, rho_sv, lambda_J*kappa_J,
   call_or_put, strike, dividend,
   tgrid, Nt, Nt-1,
   MatrixA0nm1, MatrixA1n, MatrixA2n,
   G0nm1, G1n, G2n);

  build_all_matrix(S0, sgrid, Ns,
   V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
   R0, rho_sv, lambda_J*kappa_J,
   call_or_put, strike, dividend,
   tgrid, Nt, Nt,
   MatrixA0nm1, MatrixA1nm1, MatrixA2nm1,
   G0nm1, G1nm1, G2nm1);

  // Record time independent values.
  for (i=0; i<Ns+1; i++)
{
  for (j=0; j<Nv+1; j++)
{
  GSecondMember[i][j] = G1n[i][j];
}
}


  // Terminal values
  for (i=0; i<Ns+1; i++) {
for (j=0; j<Nv+1; j++) {
  // Terminal condition
  if (call_or_put==1)
{
  U_old[i][j] = MAX(S0*exp(sgrid[i])-strike,0.);
}
  else
{
  U_old[i][j] = MAX(strike-S0*exp(sgrid[i]),0.);
}
}
  }

  // Compute the time evolution
```

```
 time_evolution(S0, sgrid, Ns,
V0, sigma_v, alpha_v, beta_v, vgrid, Nv,
R0, rho_sv,
call_or_put, strike, dividend,
pidegrid, Npide,
mu_J, sigma_J, lambda_J, kappa_J,
tgrid, Nt,
theta, scheme,
// 2-vectors
U_old, U_new, Y0, Y1, //Y2, //Y3,
G0nm1, G1nm1, G2nm1, //G3nm1, //G0n,
G1n, G2n, //G3n,
GSecondMember,
// Matrix (=3-vectors)
MatrixA0nm1, MatrixA1nm1, MatrixA2nm1, //MatrixA3nm1, //MatrixA0n,
MatrixA1n, MatrixA2n); //MatrixA3n);

  // Index in the domain for the price.
  // Find the index in sgrid corresponding to the price S0 of the asset.
  IndexS = lower_index(sgrid,Ns+1,0.);
  // Find the index in vgrid corresponding to the variance V0 of the asset.
  IndexV = lower_index(vgrid,Nv+1,V0);

  // First compute the delta (using price as temporary variable). We do an inter
  price_sd_vd = U_new[IndexS+1][IndexV];
  price_sd_vu = U_new[IndexS+1][IndexV+1];
  price_su_vd = U_new[IndexS+2][IndexV];
  price_su_vu = U_new[IndexS+2][IndexV+1];

  *ptprice=double_interpolation(price_sd_vd, price_sd_vu, price_su_vd, price_su_
sgrid[IndexS+1], sgrid[IndexS+2],
vgrid[IndexV], vgrid[IndexV+1],
sgrid[IndexS+1], V0);

  price_sd_vd = U_new[IndexS][IndexV];
  price_sd_vu = U_new[IndexS][IndexV+1];
  price_su_vd = U_new[IndexS+1][IndexV];
  price_su_vu = U_new[IndexS+1][IndexV+1];

  *ptdelta = (*ptprice - double_interpolation(price_sd_vd, price_sd_vu, price_su
  sgrid[IndexS], sgrid[IndexS+1],
```

```
  vgrid[IndexV], vgrid[IndexV+1],
  sgrid[IndexS], V0)     )
    /(S0*exp(sgrid[IndexS+1]) - S0*exp(sgrid[IndexS]));

  // Next compute the price. We do an interpolation.
  *ptprice = double_interpolation(price_sd_vd, price_sd_vu, price_su_vd, price_s
  sgrid[IndexS], sgrid[IndexS+1],
  vgrid[IndexV], vgrid[IndexV+1],
  0., V0);

  // 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(U_old[i]);
free(U_new[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]);
  }


for (i=0; i<Ns+1; i++)
  {
free(GSecondMember[i]);
  }

free(U_old);
free(U_new);
free(Y0);
free(Y1);
//free(Y2);
//free(Y3)
free(G0nm1);
free(G1nm1);
free(G2nm1);
//free(G3nm1);
//free(G0n);
free(G1n);
free(G2n);
```

```
//free(G3n);
free(GSecondMember);
  }// End of memory desallocations.
}


static int Adi_Bates(int am, double S0, NumFunc_1  *Payoff, double maturity, dou
{
  double strike;
  int call_or_put;
  double kappa_J;
  int i,Npide;

  double *sgrid, *vgrid, *tgrid, *pidegrid;
  double Smax,Vmax,Sleft,Sright,Coeff_s,Coeff_v, tau;
  int scheme;
  double theta;
  double sigma_J;

  sigma_J=sqrt(var_J);
  kappa_J = exp(mu_J+sigma_J*sigma_J/2.)-1.;
  strike = Payoff->Par[0].Val.V_PDOUBLE;

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

  Npide = Ns;
  // 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(5.*V0,1.),-5.);
  Coeff_v = Vmax/500;
  // Scheme numerical parameters.
  theta=0.5;//Theta scheme
  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));
  pidegrid=(double *)malloc((Npide+1)*sizeof(double));
  tgrid=(double *)malloc((Nt+1)*sizeof(double));

  tau = maturity/(double)Nt;
  for (i=0; i<=Nt; i++) tgrid[i] =  i * tau;
  grid_generation_spot(sgrid, Sleft, Sright, Smax, Ns, Coeff_s);
  grid_generation_variance(vgrid, Vmax, Nv, Coeff_v, V0);
  grid_generation_pide(pidegrid, 1., Npide);

  for (i=1; i<=Ns; i++)
    {
  sgrid[i] = log(sgrid[i]/S0);
    }
  sgrid[0] = 2*sgrid[1]-sgrid[2];

  ////////////////////
  // Compute price. //
  ////////////////////

  compute_price_and_delta(S0, sgrid, Ns,
  V0, sigma_v, alpha_v, beta_v,
  vgrid, Nv,
  R0, rho_sv,
  call_or_put, strike, dividend,
  pidegrid, Npide,
  mu_J, sigma_J, lambda_J, kappa_J,
  tgrid, Nt,
  theta, scheme,
  ptprice, ptdelta);

  return OK;
}

int CALC(FD_Adi_Bates)(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 Adi_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_DA
   , 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_PDOU
}

static int CHK_OPT(FD_Adi_Bates)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt
return OK;

  return WRONG;
}
#endif //PremiaCurrentVersion


static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  static int first = 1;

  if (first)
{
  Met->HelpFilenameHint = "fd_adi_bates";
  Met->Par[0].Val.V_INT2 = 100;
  Met->Par[1].Val.V_INT2 = 80;
  Met->Par[2].Val.V_INT2 = 40;
  first = 0;
}

  return OK;
}
```

```
PricingMethod MET(FD_Adi_Bates) =
{
  "FD_ADI_BATES",
  { {"Time Step", INT2, {100}, ALLOW}, {"SpaceStepNumber S", INT2, {100}, ALLOW}
{" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_Adi_Bates),
  {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID}, {" ", PRE
  CHK_OPT(FD_Adi_Bates),
  CHK_split,
  MET(Init)
};
```