Help

```
#include  "heshw1d_std.h"
#include "enums.h"
#include "error_msg.h"


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

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2015+2) //The "#els
static int CHK_OPT(FD_HybridTree)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(FD_Hout)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

// // BE CAREFUL. MUST BE 1.0 TO AVOID INTEGER DIVISION !!!
#define SPDE 1.0
#define VPDE 1.0
#define RPDE 1.0

// ONLY BOOLEAN
#define V0_IN_GRID 1

/*ZCB Data*/
static double *initial_yield; /*Values of initial_yield*/
static double *initial_forward; /*Values of initial_forward*/
static double *initial_derive_forward; /*Values of initial_derive_forward*/
static double *t_vect; /*Times used in the temporal loop of the scheme.*/
static double *tm; /*Times T of maturities read in the file initialyield.dat */
static double *Pm;/*Values of the zero coupon P(0,tm) read in the file initialyi
static char *init_tr;
static double R0_flat;
```

```
static int lecture_tr()
{
  int i;
  char ligne[30];
  char *pligne;
  double p, tt_value;
  FILE *Entrees;

  Entrees = fopen(init_tr, "r");

  if (Entrees == NULL)
    {
  printf("LE FICHIER N'A PU ETRE OUVERT. VERIFIEZ LE CHEMIN.\ n");
    }

  /* i is the number of lines that has been read */
  i = 0;
  pligne = ligne;
  Pm = (double *)malloc(200 * sizeof(double));
  tm = (double *)malloc(200 * sizeof(double));

  while (1)
    {
  pligne = fgets(ligne, sizeof(ligne), Entrees);
  if (pligne == NULL) break;
  else
        {
  sscanf(ligne, "%lf t=%lf", &p, &tt_value);
  /* The line read must be written "0.943290 t=0.5" where 0.943290 is a double f
  Pm[i] = p; /*save the price of the zero coupon*/
  tm[i] = tt_value; /*save the corresponding time*/
  i++;

        }
    }
  fclose(Entrees);

  return i;
}

static void compute_forward(double *Price, double *Time, int size, double *Forwa
```

```
{
  //Forward = - d log(P) / dt
  //Derive_Forward = d Forward / dt
  int i;

  for (i=0; i<=size-1; i++)
    {
  Forward[i] = -(log(Price[i+1])-log(Price[i]))/(Time[i+1]-Time[i]);
    }
  Forward[size] = Forward[size-1];

  for (i=0; i<=size-1; i++)
    {
  Derive_Forward[i] = (Forward[i+1]-Forward[i])/(Time[i+1]-Time[i]);
    }
  Derive_Forward[size] = Derive_Forward[size-1];
}


static void interpole(int n_price, int imax, double *t)
{
  int i, iF, j;

  n_price--;

  i = 0;
  while (t[i] <= tm[1])
    {
  initial_yield[i] = (tm[1] - t[i]) / tm[1] + t[i] * Pm[1] / tm[1];
  i++;
    }

  for (j = 0; j < n_price; j++)
    {
  while (i <= imax && t[i] < tm[j + 1] )
        {
  initial_yield[i] = (tm[j + 1] - t[i]) / (tm[j + 1] - tm[j]) * Pm[j] + (t[i] -
  i++;
        }
    }
```

```
  if (i <= imax  && t[i] > tm[n_price])
    {
  for (iF = i ; iF <= imax ; iF++)
        {
  initial_yield[iF] = Pm[n_price] + (Pm[n_price] - Pm[n_price - 1]) / (tm[n_pric
        }
    }
}


static void extract_forward(double *Forward, double *Derive_Forward, double *Tim
{
  int i;
  int j;

  i=0;
  j=0;

  while (j<=size_extract)
    {
  while ((i<=size)&&(Time[i]<t_extract[j]) )
i++;

  initial_forward[j] = Forward[i];
  initial_derive_forward[j] = Derive_Forward[i];

  j++;
    }
}


static double func_zero_coupon(int flat_flag,double* tgrid, int Nt, int time_ind
{
  // This function characterizes the mean reversion at time tgrid[time_index] su
  // dr = alpha_r ( THIS_FUNCTION(t) - r) dt + sigma_r dWt

  // Be careful, actually this function assumes that "forward" and "derive_forwa
  // computed/interpolated on t_grid...

  double b;
  double time;
```

```
  int time_index;

  b=R0_flat;
  time = tgrid[Nt]-tgrid[time_index_arg]; // T-t
  time_index = Nt-time_index_arg;


  //////////
  // Vasicek : THIS = Cst
  //return b;
  //////////

  //////////
  // Hull-White : THIS = function of time
  // An example :
  // return b+(sigma_r*sigma_r)/(2.*(alpha_r*alpha_r))*(1.-exp(-2*alpha_r*time))

  // With interest curve P(r,t,T) = exp(-b(T-t))
  // return ( b*alpha_r+(sigma_r*sigma_r)/(2.*alpha_r)*(1.-exp(-2*alpha_r*time))

  // With the forward rate f  = -d_t log(P)
  // return ( df/dt f(0,t) + alpha_r f(0,t) + (sigma_r*sigma_r)/(2.*alpha_r)*(1.


  if (flat_flag==0)
    {
  return ( b*alpha_r+(sigma_r*sigma_r)/(2.*alpha_r)*(1.-exp(-2*alpha_r*time)) )/
    }
  else
    {
  return ( (initial_derive_forward[time_index] + alpha_r * initial_forward[time_
   + (sigma_r*sigma_r)/(2.*alpha_r)*(1.-exp(-2*alpha_r*time)) )/alpha_r;
  // Take care of "/alpha_r" since mean reversion is "alpha_r * ( THIS - r)" and
    }


}

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_HHW_spot(double *sgrid, double Sleft, double Sright,
{
  int i;
  double Ximin;
  double Xiint;
```

```
  double Ximax;
  double deltaxi;

  Ximin = asinh1(-Sleft/d1);
  Xiint = (Sright-Sleft)/d1;
  Ximax = Xiint + asinh1((Smax-Sright)/d1);
  deltaxi = (Ximax-Ximin)/m1;

  // Definition of uniform grid xi.
  for (i=0; i<=m1; i++)
    {
      sgrid[i] = Ximin + i * deltaxi;
    }

  // Definition of the spot grid with the uniform grid xi.
  sgrid[0] = 0;
  for (i=1; i<=m1; i++)
{
  if (sgrid[i]<0)
{
  sgrid[i] = Sleft+d1*sinh(sgrid[i]);
}
  else
{
  if (sgrid[i]<=Xiint)
{
  sgrid[i] = Sleft+d1*sgrid[i];
}
  else
{
  sgrid[i] = Sright+d1*sinh(sgrid[i]-Xiint);
}
}
}
}

static void grid_generation_HHW_variance(double *vgrid, double Vmax, int m2, dou
{
  int j;
  double deltaeta;
```

```
  deltaeta = (asinh1(Vmax/d2))/m2;
  // Definition of uniform grid eta.
  for (j=0; j<=m2; j++)
    {
      vgrid[j] = j * deltaeta;
    }
  // Definition of the volatility grid with the uniform grid eta.
  vgrid[0] = 0;
  for (j=1; j<=m2; j++)
    {
      vgrid[j] = d2 * sinh(vgrid[j]);
    }

  if (V0_IN_GRID)
    {
      j = lower_index(vgrid, m2, c2);
      vgrid[j] = c2;
    }
}

static void grid_generation_HHW_rate(double *rgrid, double Rmax, int m3, double
{
  int k;
  double deltazeta;

  deltazeta = (asinh1((Rmax-c3)/d3)-asinh1((-Rmax-c3)/d3))/m3;

  // Definition of uniform grid eta.
  for (k=0; k<=m3; k++)
    {
      rgrid[k] = asinh1((-Rmax-c3)/d3) + k * deltazeta;
    }
  // Definition of the rate grid with the uniform grid zeta.
  for (k=0; k<=m3; k++)
    {
      rgrid[k] = c3+ d3*sinh(rgrid[k]);
    }
}

static int stencil(int i, int j, int k)
{
```

```
if ((i==0) && (j==0) && (k==0)) return 0;
if ((i==-1) && (j==0) && (k==0)) return 1;
if ((i==1) && (j==0) && (k==0)) return 2;
if ((i==0) && (j==-2) && (k==0)) return 3;
if ((i==0) && (j==-1) && (k==0)) return 4;
if ((i==0) && (j==1) && (k==0)) return 5;
if ((i==0) && (j==2) && (k==0)) return 6;
if ((i==-1) && (j==-1) && (k==0)) return 7;
if ((i==1) && (j==-1) && (k==0)) return 8;
if ((i==-1) && (j==1) && (k==0)) return 9;
if ((i==1) && (j==1) && (k==0)) return 10;
if ((i==0) && (j==0) && (k==-1)) return 11;
if ((i==-1) && (j==0) && (k==-1)) return 12;
if ((i==1) && (j==0) && (k==-1)) return 13;
if ((i==0) && (j==-1) && (k==-1)) return 14;
if ((i==0) && (j==1) && (k==-1)) return 15;
if ((i==-1) && (j==-1) && (k==-1)) return 16;
if ((i==1) && (j==-1) && (k==-1)) return 17;
if ((i==-1) && (j==1) && (k==-1)) return 18;
if ((i==1) && (j==1) && (k==-1)) return 19;
if ((i==0) && (j==0) && (k==1)) return 20;
if ((i==-1) && (j==0) && (k==1)) return 21;
if ((i==1) && (j==0) && (k==1)) return 22;
if ((i==0) && (j==-1) && (k==1)) return 23;
if ((i==0) && (j==1) && (k==1)) return 24;
if ((i==-1) && (j==-1) && (k==1)) return 25;
if ((i==1) && (j==-1) && (k==1)) return 26;
if ((i==-1) && (j==1) && (k==1)) return 27;
if ((i==1) && (j==1) && (k==1)) return 28;
/*
0,0,0 -> 0
-1,0,0 -> 1
1,0,0 -> 2
0,-2,0 -> 3
0,-1,0 -> 4
0,1,0 -> 5
0,2,0 -> 6
-1,-1,0 -> 7
1,-1,0 -> 8
-1,1,0 -> 9
1,1,0 -> 10
```

```
  0,0,-1 -> 11
  -1,0,-1 -> 12
  1,0,-1 -> 13
  0,-1,-1 -> 14
  0,1,-1 -> 15
  -1,-1,-1 -> 16
  1,-1,-1 -> 17
  -1,1,-1 -> 18
  1,1,-1 -> 19
  0,0,1 -> 20
  -1,0,1 -> 21
  1,0,1 -> 22
  0,-1,1 -> 23
  0,1,1 -> 24
  -1,-1,1 -> 25
  1,-1,1 -> 26
  -1,1,1 -> 27
  1,1,1 -> 28
  */
  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 triple_interpolation(double sd_vd_rd, double sd_vu_rd, double su_v
                                   double sd_vd_ru, double sd_vu_ru, double su_v
                                   double sd, double su, double vd, double vu, d
                                   double s, double v, double r)
{
  double v_sd_rd,v_su_rd,v_s_rd,v_sd_ru,v_su_ru,v_s_ru;

  v_sd_rd = interpolation (vd, sd_vd_rd, vu, sd_vu_rd, v);
  v_su_rd = interpolation (vd, su_vd_rd, vu, su_vu_rd, v);
  v_s_rd = interpolation (sd, v_sd_rd, su, v_su_rd, s);
```

```
  v_sd_ru = interpolation (vd, sd_vd_ru, vu, sd_vu_ru, v);
  v_su_ru = interpolation (vd, su_vd_ru, vu, su_vu_ru, v);
  v_s_ru = interpolation (sd, v_sd_ru, su, v_su_ru, s);

  return interpolation (rd, v_s_rd, ru, v_s_ru, r);
}

static int it_exists_stencil(int i, int Ns, int j, int Nv, int k, int Nr, int st
{
  // We use i from 1 to Ns, j from 0 to Nv-1 and k from 0 to Nr
  // We use points i-1 -> i+1, j-2 -> j+2 and k-1 -> k+1

  /*
  0,0,0 -> 0
  -1,0,0 -> 1
  1,0,0 -> 2
  0,-2,0 -> 3
  0,-1,0 -> 4
  0,1,0 -> 5
  0,2,0 -> 6
  -1,-1,0 -> 7
  1,-1,0 -> 8
  -1,1,0 -> 9
  1,1,0 -> 10
  0,0,-1 -> 11
  -1,0,-1 -> 12
  1,0,-1 -> 13
  0,-1,-1 -> 14
  0,1,-1 -> 15
  -1,-1,-1 -> 16
  1,-1,-1 -> 17
  -1,1,-1 -> 18
  1,1,-1 -> 19
  0,0,1 -> 20
  -1,0,1 -> 21
  1,0,1 -> 22
  0,-1,1 -> 23
  0,1,1 -> 24
  -1,-1,1 -> 25
  1,-1,1 -> 26
  -1,1,1 -> 27
```

```
  1,1,1 -> 28
  */

  if ((i>1) && (i<Ns)&& (j>1) && (j<Nv-2) && (k>0) && (k<Nr)) // Strict interior
    {
  return 1;
    }

  if (i==1)
if ((stencil== 1) || (stencil== 7) || (stencil== 9) || (stencil==12) || (stencil
|| (stencil==18) || (stencil==21) || (stencil==25) || (stencil==27))
  return 0;

  if (i==Ns)
if ((stencil== 2) || (stencil== 8) || (stencil==10) || (stencil==13) || (stencil
|| (stencil==19) || (stencil==22) || (stencil==26) || (stencil==28))
  return 0;

  if (j==0)
if ((stencil== 3) || (stencil== 4) || (stencil==7) || (stencil== 8) || (stencil=
|| (stencil==16) || (stencil==17) || (stencil==23) || (stencil==25) || (stencil=
  return 0;

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

  if (j==Nv-2)
if (stencil== 6)
  return 0;

  if (j==Nv-1)
if ((stencil== 5) || (stencil== 6) || (stencil==9) || (stencil== 10) || (stencil
|| (stencil==18) || (stencil==19) || (stencil==24) || (stencil==27) || (stencil=
  return false;

  if (k==0)
if ((stencil==11) || (stencil==12) || (stencil==13) || (stencil==14) || (stencil
|| (stencil==16) || (stencil==17) || (stencil==18) || (stencil==19))
  return 0;
```

```
  if (k==Nr)
if ((stencil==20) || (stencil==21) || (stencil==22) || (stencil==23) || (stencil
|| (stencil==25) || (stencil==26) || (stencil==27) || (stencil==28))
  return 0;

  return 1;
}

static void point_of_stencil(int i, int j, int k, int stencil, int* pi, int* pj,
{
  if (stencil==0)  {   *pi=i;     *pj=j;      *pk=k;}
  if (stencil==1)  {   *pi=i-1;   *pj=j;      *pk=k;}
  if (stencil==2)  {   *pi=i+1;   *pj=j;      *pk=k;}
  if (stencil==3)  {   *pi=i;     *pj=j-2;    *pk=k;}
  if (stencil==4)  {   *pi=i;     *pj=j-1;    *pk=k;}
  if (stencil==5)  {   *pi=i;     *pj=j+1;    *pk=k;}
  if (stencil==6)  {   *pi=i;     *pj=j+2;    *pk=k;}
  if (stencil==7)  {   *pi=i-1;   *pj=j-1;    *pk=k;}
  if (stencil==8)  {   *pi=i+1;   *pj=j-1;    *pk=k;}
  if (stencil==9)  {   *pi=i-1;   *pj=j+1;    *pk=k;}
  if (stencil==10) {   *pi=i+1;   *pj=j+1;    *pk=k;}
  if (stencil==11) {   *pi=i;     *pj=j;      *pk=k-1;}
  if (stencil==12) {   *pi=i-1;   *pj=j;      *pk=k-1;}
  if (stencil==13) {   *pi=i+1;   *pj=j;      *pk=k-1;}
  if (stencil==14) {   *pi=i;     *pj=j-1;    *pk=k-1;}
  if (stencil==15) {   *pi=i;     *pj=j+1;    *pk=k-1;}
  if (stencil==16) {   *pi=i-1;   *pj=j-1;    *pk=k-1;}
  if (stencil==17) {   *pi=i+1;   *pj=j-1;    *pk=k-1;}
  if (stencil==18) {   *pi=i-1;   *pj=j+1;    *pk=k-1;}
  if (stencil==19) {   *pi=i+1;   *pj=j+1;    *pk=k-1;}
  if (stencil==20) {   *pi=i;     *pj=j;      *pk=k+1;}
  if (stencil==21) {   *pi=i-1;   *pj=j;      *pk=k+1;}
  if (stencil==22) {   *pi=i+1;   *pj=j;      *pk=k+1;}
  if (stencil==23) {   *pi=i;     *pj=j-1;    *pk=k+1;}
  if (stencil==24) {   *pi=i;     *pj=j+1;    *pk=k+1;}
  if (stencil==25) {   *pi=i-1;   *pj=j-1;    *pk=k+1;}
  if (stencil==26) {   *pi=i+1;   *pj=j-1;    *pk=k+1;}
  if (stencil==27) {   *pi=i-1;   *pj=j+1;    *pk=k+1;}
  if (stencil==28) {   *pi=i+1;   *pj=j+1;    *pk=k+1;}
}
```

```
static double bc_spot_min(double strike, double divid,
                          double* tgrid, int Nt, int time_index,
                          double sigma_v, double alpha_v, double beta_v,
                          double sigma_r, double alpha_r,
                          double rho_sv, double rho_sr, double rho_vr,
                          double *sgrid, int i, double *vgrid, int j, double *rg
                          int call_or_put, int am)
{
  double time;
  time = tgrid[time_index];

  if (call_or_put==0) // Bond -> Spot min useless.
return 0.;
  if (call_or_put==1) // Call
// Dirichlet = 0
return 0.;
  else // (call_or_put==-1) // Put
// Dirichlet = strike*exp(-r*t)
return strike*exp(-rgrid[k]*time);
}

static double bc_spot_max(double strike, double divid,
                          double* tgrid, int Nt, int time_index,
                          double sigma_v, double alpha_v, double beta_v,
                          double sigma_r, double alpha_r,
                          double rho_sv, double rho_sr, double rho_vr,
                          double *sgrid, int i, double *vgrid, int j, double *rg
                          int call_or_put, int am)
{
  double time;
  time = tgrid[time_index];

  if (call_or_put==0) // Bond -> Spot max useless.
return 0.;
  if (call_or_put==1) // Call
// Neumann = exp(-dt)
return exp(-divid*time) * (sgrid[i]-sgrid[i-1]);
  else // (call_or_put==-1) // Put
// Neumann = 0
```

```
return 0.;
}


static double bc_var_max(double strike, double divid,
                         double* tgrid, int Nt, int time_index,
                         double sigma_v, double alpha_v, double beta_v,
                         double sigma_r, double alpha_r,
                         double rho_sv, double rho_sr, double rho_vr,
                         double *sgrid, int i, double *vgrid, int j, double *rgr
                         int call_or_put, int am)
{
  double time;
  time = tgrid[time_index];

  if (call_or_put==0) // Bond -> Var max useless.
return 0.;
  if (call_or_put==1) // Call
// Dirichlet = S * exp(-d*t)
return sgrid[i]*exp(-divid*time);
  else // (call_or_put==-1) // Put
// Dirichlet = strike*exp(-r*t)
return strike*exp(-rgrid[k]*time);
}

static double bc_rate_min(int flat_flag,double strike, double divid,
                          double* tgrid, int Nt, int time_index,
                          double sigma_v, double alpha_v, double beta_v,
                          double sigma_r, double alpha_r,
                          double rho_sv, double rho_sr, double rho_vr,
                          double *sgrid, int i, double *vgrid, int j, double *rg
                          int call_or_put, int am)
{
  double time;
  double b;
  double a;
  double deriv;
  int decalage_derivative;


  time = tgrid[time_index];
```

```
  b=func_zero_coupon(flat_flag,tgrid, Nt, time_index, alpha_r, sigma_r);
  a=alpha_r;
  deriv = (exp(-a*time)-1.)/a; // Derivative of func(r) in the exponential. Take
  decalage_derivative = 1; // 0 or 1. Use Neumann condition at point Rmin or Rmi

  if (call_or_put==0) // Bond
    {
  if (flat_flag==0)
        {
  // Assuming Vasicek model, the closed formula gives the value of the derivativ
  // Take care of the sign minus in the return value since U0 = U1 - Neumann * D
  return -1. * deriv*exp(-b*time + (b-rgrid[k+decalage_derivative])/a*(1.-exp(-a
 + sigma_r*sigma_r/(2.*a*a)*time+sigma_r*sigma_r/(4.*a*a*a)*(1.-exp(-2.*a*time))
 - sigma_r*sigma_r/(a*a*a)*(1.-exp(-a*time)))
* (rgrid[k+1]-rgrid[k]);
        }
  else
        {
  // Assuming Hull-White model,
  // Bond = Price(0,t)/Price(0,0) * exp(-deriv * forward(0,t) - sigma*sigma (1-e
  // * exp (deriv * R)
  // Take care of the sign minus in the return value since U0 = U1 - Neumann * D
  // So "d Bond / dr" = -deriv * Price(0,t)/Price(0,0) * exp(-deriv * forward(0,


  // Furtermore, in all cases, homogeneous Neumann should be enough...
  return 0.;
        }

    }
  if (call_or_put==1) // Call
// Neumann = 0
return 0.;
  else // (call_or_put==-1) // Put
// Neumann = 0
return 0.;
}

static double bc_rate_max(int flat_flag,double strike, double divid,
                          double* tgrid, int Nt, int time_index,
                          double sigma_v, double alpha_v, double beta_v,
```

```
                          double sigma_r, double alpha_r,
                          double rho_sv, double rho_sr, double rho_vr,
                          double *sgrid, int i, double *vgrid, int j, double *rg
                          int call_or_put, int am)
{
  double time;
  double b;
  double a;
  double deriv;
  int decalage_derivative;

  time = tgrid[time_index];
  b=func_zero_coupon(flat_flag,tgrid, Nt, time_index, alpha_r, sigma_r);
  a=alpha_r;
  deriv = (exp(-a*time)-1.)/a; // Derivative of func(r) in the exponential. Take
  decalage_derivative = 0; // 0 or 1. Use Neumann condition at point Rmax or Rma


  if (call_or_put==0) // Bond
    {
  if (flat_flag==0)
        {
  // Assuming Vasicek model, the closed formula gives the value of the derivativ
  // Take care of the sign plus in the return value since UN = UN-1 + Neumann *
  return deriv*exp(-b*time + (b-rgrid[k-decalage_derivative])/a*(1.-exp(-a*time)
   + sigma_r*sigma_r/(2.*a*a)*time+sigma_r*sigma_r/(4.*a*a*a)*(1.-exp(-2.*a*time
   - sigma_r*sigma_r/(a*a*a)*(1.-exp(-a*time)))
* (rgrid[k]-rgrid[k-1]);
        }
  else
        {
  // Assuming Hull-White model,
  // Bond = Price(0,t)/Price(0,0) * exp(-deriv * forward(0,t) - sigma*sigma (1-e
  // * exp (deriv * R)
  // So "d Bond / dr" = deriv * Price(0,t)/Price(0,0) * exp(-deriv * forward(0,t

  // Furthermore, in all cases, homogeneous Neumann should be enough...
  return 0.;
        }
    }
  if (call_or_put==1) // Call
```

```
// Neumann = 0
return 0.;
  else // (call_or_put==-1) // Put
// Neumann = 0
return 0.;
}



static void build_all_matrix(int flat_flag,double strike, double divid,
                             double* tgrid, int Nt, int time_index,
                             double sigma_v, double alpha_v, double beta_v,
                             double sigma_r, double alpha_r,
                             double rho_sv, double rho_sr, double rho_vr,
                             double *sgrid, int Ns, double *vgrid, int Nv, doubl
                             int call_or_put, int am,
                             double ****MatrixA0, double ****MatrixA1, double **
                             double ***G0, double ***G1, double ***G2, double **
{
  double actualspoint;
  double actualvpoint;
  double actualrpoint;

  double Dsi, Dsip1;
  double Dvjm1, Dvj, Dvjp1, Dvjp2;
  double Drk, Drkp1;

  double convection_s, diffusion_s;
  double convection_v, diffusion_v;
  double convection_r, diffusion_r;
  double order_0, mixted_sv, mixted_sr, mixted_vr;

  double *cs;
  double *ds;
  double *cv;
  double *dv;
  double *cr;
  double *dr;
  double *msv, *msr, *mvr;

  // 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;

//    double ralpha_km2, ralpha_km1, ralpha_k0;
double rbeta_km1, rbeta_k0, rbeta_kp1;
//    double rgamma_k0, rgamma_kp1, rgamma_kp2;
//    double rdelta_km1, rdelta_k0, rdelta_kp1;

int i, j, k, st;

double G_cs, G_ds, G_cv, G_dv, G_cr, G_dr, G_msv, G_msr, G_mvr;

cs=(double*)malloc(29*sizeof(double));
ds=(double*)malloc(29*sizeof(double));
cv=(double*)malloc(29*sizeof(double));
dv=(double*)malloc(29*sizeof(double));
cr=(double*)malloc(29*sizeof(double));
dr=(double*)malloc(29*sizeof(double));
msv=(double*)malloc(29*sizeof(double));
msr=(double*)malloc(29*sizeof(double));
mvr=(double*)malloc(29*sizeof(double));


for(k=0;k<Nr+1;k++)
  {
for(j=0;j<Nv;j++)
      {
for(i=1;i<Ns+1;i++)
          {
for(st=0;st<29;st++)
              {
cs[st]=0.;
ds[st]=0.;
cv[st]=0.;
```

```
dv[st]=0.;
cr[st]=0.;
dr[st]=0.;
msv[st]=0.;
msr[st]=0.;
mvr[st]=0.;
              }
G_cs=0.;
G_ds=0.;
G_cv=0.;
G_dv=0.;
G_cr=0.;
G_dr=0.;
G_msv=0.;
G_msr=0.;
G_mvr=0.;

actualspoint = sgrid[i];
actualvpoint = vgrid[j];
actualrpoint = rgrid[k];

convection_s = (actualrpoint-divid) * actualspoint;
diffusion_s = actualspoint * actualspoint * actualvpoint/2.0;
convection_v = alpha_v*(beta_v - actualvpoint); //   convection_v = kappa*(eta
diffusion_v = sigma_v * sigma_v * actualvpoint/2.0;
convection_r = alpha_r*(func_zero_coupon(flat_flag,tgrid, Nt, time_index, alph
diffusion_r = sigma_r * sigma_r/2.0;

order_0 = -actualrpoint;
mixted_sv = rho_sv * sigma_v * actualspoint * actualvpoint;
mixted_sr = rho_sr * sigma_r * actualspoint * sqrt(actualvpoint);
mixted_vr = rho_vr * sigma_v * sigma_r * sqrt(actualvpoint);


convection_s = convection_s * SPDE;
diffusion_s = diffusion_s * SPDE;
convection_v = convection_v * VPDE;
diffusion_v = diffusion_v * VPDE;
convection_r = convection_r * RPDE;
diffusion_r = diffusion_r * RPDE;
```

```
  mixted_sv = mixted_sv * SPDE * VPDE;
  mixted_sr = mixted_sr * SPDE * RPDE;
  mixted_vr = mixted_vr * VPDE * RPDE;



  ////////////////////
  // Diffusion and Convection S
  ////////////////////
  {
if (i==1) // S~Smin -> Dirichlet
  {
Dsi = sgrid[i]-sgrid[i-1];
Dsip1 = sgrid[i+1]-sgrid[i];
//ds[stencil(-1,0,0)]=2.0/(Dsi*(Dsi+Dsip1));
G_ds += 2.0/(Dsi*(Dsi+Dsip1)) * bc_spot_min(strike, divid,
tgrid, Nt, time_index,
sigma_v, alpha_v, beta_v,
sigma_r, alpha_r,
rho_sv, rho_sr, rho_vr,
sgrid, i-1, vgrid, j, rgrid, k,
call_or_put, am);
ds[stencil(0,0,0)]=-2.0/(Dsi*Dsip1);
ds[stencil(1,0,0)]=2.0/(Dsip1*(Dsi+Dsip1));

//cs[stencil(-1,0,0)]=-Dsip1/(Dsi*(Dsi+Dsip1));
G_cs += -Dsip1/(Dsi*(Dsi+Dsip1)) * bc_spot_min(strike, divid,
   tgrid, Nt, time_index,
   sigma_v, alpha_v, beta_v,
   sigma_r, alpha_r,
   rho_sv, rho_sr, rho_vr,
   sgrid, i-1, vgrid, j, rgrid, k,
   call_or_put, am);
cs[stencil(0,0,0)]=(Dsip1-Dsi)/(Dsi*Dsip1);
cs[stencil(1,0,0)]=Dsi/(Dsip1*(Dsi+Dsip1));
  }
else
  {
if (i==Ns) // S=Smax -> Neumann
  {
Dsi = sgrid[i]-sgrid[i-1];
```

```
Dsip1 = Dsi;
ds[stencil(-1,0,0)]=2.0/(Dsi*(Dsi+Dsip1));
ds[stencil(0,0,0)]=-2.0/(Dsi*Dsip1) + 2.0/(Dsip1*(Dsi+Dsip1));
//ds[stencil(1,0,0)]=2.0/(Dsip1*(Dsi+Dsip1));
G_ds = 2.0/(Dsip1*(Dsi+Dsip1)) * bc_spot_max(strike, divid,
 tgrid, Nt, time_index,
 sigma_v, alpha_v, beta_v,
 sigma_r, alpha_r,
 rho_sv, rho_sr, rho_vr,
 sgrid, i, vgrid, j, rgrid, k,
 call_or_put, am);

cs[stencil(-1,0,0)]=-Dsip1/(Dsi*(Dsi+Dsip1));
cs[stencil(0,0,0)]=(Dsip1-Dsi)/(Dsi*Dsip1) + Dsi/(Dsip1*(Dsi+Dsip1));
///HERE
///MODIFICATION
//cs[stencil(1,0,0)]=Dsi/(Dsip1*(Dsi+Dsip1));
G_cs += Dsi/(Dsip1*(Dsi+Dsip1)) * bc_spot_max(strike, divid,
  tgrid, Nt, time_index,
  sigma_v, alpha_v, beta_v,
  sigma_r, alpha_r,
  rho_sv, rho_sr, rho_vr,
  sgrid, i, vgrid, j, rgrid, k,
  call_or_put, am);
  }
else
  {
Dsi = sgrid[i]-sgrid[i-1];
Dsip1 = sgrid[i+1]-sgrid[i];
ds[stencil(-1,0,0)]=2.0/(Dsi*(Dsi+Dsip1));
ds[stencil(0,0,0)]=-2.0/(Dsi*Dsip1);
ds[stencil(1,0,0)]=2.0/(Dsip1*(Dsi+Dsip1));

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

```
  ////////////////////
  {
if (j==0) // V=Vmin -> Forward
  {
// if (j==0) // V=Vmin -> Diff = 0

Dvjp1 = vgrid[j+1]-vgrid[j];
Dvjp2 = vgrid[j+2]-vgrid[j+1];
cv[stencil(0,0,0)]=-(2.0*Dvjp1+Dvjp2)/(Dvjp1*(Dvjp1+Dvjp2));
cv[stencil(0,1,0)]=(Dvjp1+Dvjp2)/(Dvjp1*Dvjp2);
cv[stencil(0,2,0)]=-Dvjp1/(Dvjp2*(Dvjp1+Dvjp2));
  }
else
  {
if (j==Nv-1) // V~Vmax -> Backward
  {
// if (j==Nv-1) // V=Vmax -> Dirichlet
Dvjm1 = vgrid[j-1]-vgrid[j-2];
Dvj = vgrid[j]-vgrid[j-1];
Dvjp1 = vgrid[j+1]-vgrid[j];
dv[stencil(0,-1,0)]=2.0/(Dvj*(Dvj+Dvjp1));
dv[stencil(0,0,0)]=-2.0/(Dvj*Dvjp1);
//dv[stencil(0,1,0)]=2.0/(Dvjp1*(Dvj+Dvjp1));
G_dv += 2.0/(Dvjp1*(Dvj+Dvjp1)) * bc_var_max(strike, divid,
 tgrid, Nt, time_index,
 sigma_v, alpha_v, beta_v,
 sigma_r, alpha_r,
 rho_sv, rho_sr, rho_vr,
 sgrid, i, vgrid, j+1, rgrid, k,
 call_or_put, am);

cv[stencil(0,0,0)]=(Dvjm1+2.0*Dvj)/(Dvj*(Dvjm1+Dvj));
cv[stencil(0,-1,0)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj);
cv[stencil(0,-2,0)]=Dvj/(Dvjm1*(Dvjm1+Dvj));
  }
else
  {
//Dvjm1 = vgrid[j-1]-vgrid[j-2];
if (j>1)
  {
Dvjm1 = vgrid[j-1]-vgrid[j-2];
```

```
  }
else
  {
Dvjm1 = vgrid[j]-vgrid[j-1];
  }


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

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

if ((convection_v<0) && (j>1)) // vol>beta and j>1 -> Backward
  {
cv[stencil(0,0,0)]=(Dvjm1+2.0*Dvj)/(Dvj*(Dvjm1+Dvj));
cv[stencil(0,-1,0)]=-(Dvjm1+Dvj)/(Dvjm1*Dvj);
cv[stencil(0,-2,0)]=Dvj/(Dvjm1*(Dvjm1+Dvj));
  }
else // vol<= beta or j==1 -> central
  {
cv[stencil(0,-1,0)]=-Dvjp1/(Dvj*(Dvj+Dvjp1));
cv[stencil(0,0,0)]=(Dvjp1-Dvj)/(Dvj*Dvjp1);
cv[stencil(0,1,0)]=Dvj/(Dvjp1*(Dvj+Dvjp1));
  }
  }
  }
  }
  ////////////////////
  // Diffusion and convection R
  ////////////////////
  {
if (k==0) // R=Rmin -> Neumann
  {
Drkp1 = rgrid[k+1]-rgrid[k];
Drk = Drkp1;

//dr[stencil(0,0,-1)]=2.0/(Drk*(Drk+Drkp1));
G_dr += 2.0/(Drk*(Drk+Drkp1)) * bc_rate_min(flat_flag,strike, divid,
tgrid, Nt, time_index,
```

```
sigma_v, alpha_v, beta_v,
sigma_r, alpha_r,
rho_sv, rho_sr, rho_vr,
sgrid, i, vgrid, j, rgrid, k,
call_or_put, am);
dr[stencil(0,0,0)]=-2.0/(Drk*Drkp1)+2.0/(Drk*(Drk+Drkp1));
dr[stencil(0,0,1)]=2.0/(Drkp1*(Drk+Drkp1));

//cr[stencil(0,0,-1)]=-Drkp1/(Drk*(Drk+Drkp1));
G_cr += -Drkp1/(Drk*(Drk+Drkp1)) * bc_rate_min(flat_flag,strike, divid,
   tgrid, Nt, time_index,
   sigma_v, alpha_v, beta_v,
   sigma_r, alpha_r,
   rho_sv, rho_sr, rho_vr,
   sgrid, i, vgrid, j, rgrid, k,
   call_or_put, am);
cr[stencil(0,0,0)]=(Drkp1-Drk)/(Drk*Drkp1)-Drkp1/(Drk*(Drk+Drkp1)); // Correctio
cr[stencil(0,0,1)]=Drk/(Drkp1*(Drk+Drkp1));
  }
else
  {
if (k==Nr) // R=Rmax -> Neumann
  {
Drk = rgrid[k]-rgrid[k-1];
Drkp1 = Drk;

dr[stencil(0,0,-1)]=2.0/(Drk*(Drk+Drkp1));
dr[stencil(0,0,0)]=-2.0/(Drk*Drkp1) + 2.0/(Drkp1*(Drk+Drkp1));
//dr[stencil(0,0,1)]=2.0/(Drkp1*(Drk+Drkp1));
G_dr += 2.0/(Drkp1*(Drk+Drkp1)) * bc_rate_max(flat_flag,strike, divid,
  tgrid, Nt, time_index,
  sigma_v, alpha_v, beta_v,
  sigma_r, alpha_r,
  rho_sv, rho_sr, rho_vr,
  sgrid, i, vgrid, j, rgrid, k,
  call_or_put, am);

cr[stencil(0,0,-1)]=-Drkp1/(Drk*(Drk+Drkp1));
cr[stencil(0,0,0)]=(Drkp1-Drk)/(Drk*Drkp1) + Drk/(Drkp1*(Drk+Drkp1)); // Correct
//cr[stencil(0,0,1)]=Drk/(Drkp1*(Drk+Drkp1));
G_cr += Drk/(Drkp1*(Drk+Drkp1)) * bc_rate_max(flat_flag,strike, divid,
```

```
  tgrid, Nt, time_index,
  sigma_v, alpha_v, beta_v,
  sigma_r, alpha_r,
  rho_sv, rho_sr, rho_vr,
  sgrid, i, vgrid, j, rgrid, k,
  call_or_put, am);
  }
else
  {
Drk = rgrid[k]-rgrid[k-1];
Drkp1 = rgrid[k+1]-rgrid[k];

dr[stencil(0,0,-1)]=2.0/(Drk*(Drk+Drkp1));
dr[stencil(0,0,0)]=-2.0/(Drk*Drkp1);
dr[stencil(0,0,1)]=2.0/(Drkp1*(Drk+Drkp1));

cr[stencil(0,0,-1)]=-Drkp1/(Drk*(Drk+Drkp1));
cr[stencil(0,0,0)]=(Drkp1-Drk)/(Drk*Drkp1); // Correction done : Suppress a minu
cr[stencil(0,0,1)]=Drk/(Drkp1*(Drk+Drkp1));

  }
  }
  }

  // Mixted SV
  {
// V=Vmin or Vmax or S=Smax or Smin -->> Mixted SV = 0
//if (i==0) // mx_sv = 0 by s=0 and Dirichlet (Not in the computational domain)
//if (i==Ns) // mx_sv = 0 by Neumann
//if (j==0) // mx_sv = 0 by v=0
//if (j==Nv) // mx_sv = 0 by Dirichlet (Not in the computational domain)
if ((i>1) && (i<Ns) && (0<j) && (j<Nv-1)) // Strict interior 1<i<Ns and 0<j<Nv-1
  {
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(0,0,0)] = sbeta_i0 * vbeta_j0;
msv[stencil(-1,0,0)] = sbeta_im1 * vbeta_j0;
msv[stencil(1,0,0)] = sbeta_ip1 * vbeta_j0;
msv[stencil(0,-1,0)] = sbeta_i0 * vbeta_jm1;
msv[stencil(0,1,0)] = sbeta_i0 * vbeta_jp1;
msv[stencil(-1,-1,0)] = sbeta_im1 * vbeta_jm1;
msv[stencil(1,-1,0)] = sbeta_ip1 * vbeta_jm1;
msv[stencil(-1,1,0)] = sbeta_im1 * vbeta_jp1;
msv[stencil(1,1,0)] = sbeta_ip1 * vbeta_jp1;
  }
else // i==1 or i==Ns or j==0 or j==Nv-1
  {
if ((0<j) && (i<Ns)) // Otherwise mx_sv = 0. So now i==1 (with j>0) or j==Nv-1 (
  {
if (i==1)
  {
if (j==Nv-1) // i==1 and j==Nv-1 -->> Double Dirichlet condition at point (i-1,j
  // It should be compatible.
  {
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(0,0,0)] = sbeta_i0 * vbeta_j0;
//msv[stencil(-1,0,0)] = sbeta_im1 * vbeta_j0;
G_msv += sbeta_im1 * vbeta_j0 * bc_spot_min(strike, divid,
tgrid, Nt, time_index,
sigma_v, alpha_v, beta_v,
sigma_r, alpha_r,
```

```
rho_sv, rho_sr, rho_vr,
sgrid, i-1, vgrid, j, rgrid, k,
call_or_put, am);
msv[stencil(1,0,0)] = sbeta_ip1 * vbeta_j0;
msv[stencil(0,-1,0)] = sbeta_i0 * vbeta_jm1;
//msv[stencil(0,1,0)] = sbeta_i0 * vbeta_jp1;
G_msv += sbeta_i0 * vbeta_jp1 * bc_var_max(strike, divid,
   tgrid, Nt, time_index,
   sigma_v, alpha_v, beta_v,
   sigma_r, alpha_r,
   rho_sv, rho_sr, rho_vr,
   sgrid, i, vgrid, j+1, rgrid, k,
   call_or_put, am);
//msv[stencil(-1,-1,0)] = sbeta_im1 * vbeta_jm1;
G_msv += sbeta_im1 * vbeta_jm1 * bc_spot_min(strike, divid,
 tgrid, Nt, time_index,
 sigma_v, alpha_v, beta_v,
 sigma_r, alpha_r,
 rho_sv, rho_sr, rho_vr,
 sgrid, i-1, vgrid, j-1, rgrid, k,
 call_or_put, am);
msv[stencil(1,-1,0)] = sbeta_ip1 * vbeta_jm1;

// Here double condition
//msv[stencil(-1,1,0)] = sbeta_im1 * vbeta_jp1;
G_msv += sbeta_im1 * vbeta_jp1 * bc_spot_min(strike, divid,
 tgrid, Nt, time_index,
 sigma_v, alpha_v, beta_v,
 sigma_r, alpha_r,
 rho_sv, rho_sr, rho_vr,
 sgrid, i-1, vgrid, j+1, rgrid, k,
 call_or_put, am);
// The previous line shoule be equivalent to the next line.
//G_msv += sbeta_im1 * vbeta_jp1 * bc_var_max(divid, time,
//sigma_v, alpha_v, beta_v,
//sigma_r, alpha_r,
//rho_sv, rho_sr, rho_vr,
//sgrid, i-1, vgrid, j+1, rgrid, k,
//call_or_put, am);

//msv[stencil(1,1,0)] = sbeta_ip1 * vbeta_jp1;
```

```
G_msv += sbeta_ip1 * vbeta_jp1 * bc_var_max(strike, divid,
tgrid, Nt, time_index,
sigma_v, alpha_v, beta_v,
sigma_r, alpha_r,
rho_sv, rho_sr, rho_vr,
sgrid, i+1, vgrid, j+1, rgrid, k,
call_or_put, am);
  }
else // i==1 and 0<j<Nv-1
  {
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(0,0,0)] = sbeta_i0 * vbeta_j0;
//msv[stencil(-1,0,0)] = sbeta_im1 * vbeta_j0;
G_msv += sbeta_im1 * vbeta_j0 * bc_spot_min(strike, divid,
tgrid, Nt, time_index,
sigma_v, alpha_v, beta_v,
sigma_r, alpha_r,
rho_sv, rho_sr, rho_vr,
sgrid, i-1, vgrid, j, rgrid, k,
call_or_put, am);
msv[stencil(1,0,0)] = sbeta_ip1 * vbeta_j0;
msv[stencil(0,-1,0)] = sbeta_i0 * vbeta_jm1;
msv[stencil(0,1,0)] = sbeta_i0 * vbeta_jp1;
//msv[stencil(-1,-1,0)] = sbeta_im1 * vbeta_jm1;
G_msv += sbeta_im1 * vbeta_jm1 * bc_spot_min(strike, divid,
 tgrid, Nt, time_index,
 sigma_v, alpha_v, beta_v,
 sigma_r, alpha_r,
 rho_sv, rho_sr, rho_vr,
 sgrid, i-1, vgrid, j-1, rgrid, k,
```

```
 call_or_put, am);
msv[stencil(1,-1,0)] = sbeta_ip1 * vbeta_jm1;
//msv[stencil(-1,1,0)] = sbeta_im1 * vbeta_jp1;
G_msv += sbeta_im1 * vbeta_jp1 * bc_spot_min(strike, divid,
 tgrid, Nt, time_index,
 sigma_v, alpha_v, beta_v,
 sigma_r, alpha_r,
 rho_sv, rho_sr, rho_vr,
 sgrid, i-1, vgrid, j+1, rgrid, k,
 call_or_put, am);
msv[stencil(1,1,0)] = sbeta_ip1 * vbeta_jp1;
  }
  }
else // 1<i<Ns and j==Nv-1
  {
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(0,0,0)] = sbeta_i0 * vbeta_j0;
msv[stencil(-1,0,0)] = sbeta_im1 * vbeta_j0;
msv[stencil(1,0,0)] = sbeta_ip1 * vbeta_j0;
msv[stencil(0,-1,0)] = sbeta_i0 * vbeta_jm1;
//msv[stencil(0,1,0)] = sbeta_i0 * vbeta_jp1;
G_msv += sbeta_i0 * vbeta_jp1 * bc_var_max(strike, divid,
   tgrid, Nt, time_index,
   sigma_v, alpha_v, beta_v,
   sigma_r, alpha_r,
   rho_sv, rho_sr, rho_vr,
   sgrid, i, vgrid, j+1, rgrid, k,
   call_or_put, am);
msv[stencil(-1,-1,0)] = sbeta_im1 * vbeta_jm1;
msv[stencil(1,-1,0)] = sbeta_ip1 * vbeta_jm1;
```

```
//msv[stencil(-1,1,0)] = sbeta_im1 * vbeta_jp1;
G_msv += sbeta_im1 * vbeta_jp1 * bc_var_max(strike, divid,
tgrid, Nt, time_index,
sigma_v, alpha_v, beta_v,
sigma_r, alpha_r,
rho_sv, rho_sr, rho_vr,
sgrid, i-1, vgrid, j+1, rgrid, k,
call_or_put, am);
//msv[stencil(1,1,0)] = sbeta_ip1 * vbeta_jp1;
G_msv += sbeta_ip1 * vbeta_jp1 * bc_var_max(strike, divid,
tgrid, Nt, time_index,
sigma_v, alpha_v, beta_v,
sigma_r, alpha_r,
rho_sv, rho_sr, rho_vr,
sgrid, i+1, vgrid, j+1, rgrid, k,
call_or_put, am);
  }
  }
  }
  }
  // Mixted SR
  {
// S=Smax or Smin or R=Rmin or Rmax -->> Mixted SR = 0
//if (i==0) // mx_sr = 0 by s=0 and Dirichlet (Not in the computational domain)
//if (i==Ns) // mx_sr = 0 by Neumann
//if (k==0) // mx_sr = 0 by Neumann
//if (k==Nr) // mx_sr = 0 by Neumann
if ((i>1) && (i<Ns) && (0<k) && (k<Nr)) // Strict interior 1<i<Ns and 0<k<Nr
  {
Dsi = sgrid[i]-sgrid[i-1];
Dsip1 = sgrid[i+1]-sgrid[i];
Drk = rgrid[k]-rgrid[k-1];
Drkp1 = rgrid[k+1]-rgrid[k];

sbeta_im1 = -Dsip1/(Dsi*(Dsi+Dsip1));
sbeta_i0 = (Dsip1-Dsi)/(Dsi*Dsip1);
sbeta_ip1 = Dsi/(Dsip1*(Dsi+Dsip1));
rbeta_km1 = -Drkp1/(Drk*(Drk+Drkp1));
rbeta_k0 = (Drkp1-Drk)/(Drk*Drkp1);
rbeta_kp1 = Drk/(Drkp1*(Drk+Drkp1));
```

```
msr[stencil(0,0,0)] = sbeta_i0 * rbeta_k0;
msr[stencil(-1,0,0)] = sbeta_im1 * rbeta_k0;
msr[stencil(1,0,0)] = sbeta_ip1 * rbeta_k0;
msr[stencil(0,0,-1)] = sbeta_i0 * rbeta_km1;
msr[stencil(0,0,1)] = sbeta_i0 * rbeta_kp1;
msr[stencil(-1,0,-1)] = sbeta_im1 * rbeta_km1;
msr[stencil(1,0,-1)] = sbeta_ip1 * rbeta_km1;
msr[stencil(-1,0,1)] = sbeta_im1 * rbeta_kp1;
msr[stencil(1,0,1)] = sbeta_ip1 * rbeta_kp1;
  }
else // i==1 or i==Ns or k==0 or k==Nr
  {
if ((0<k) && (k<Nr) && (i<Ns)) // Otherwise mx_sr = 0. So now i==1 (with 0<k<Nr)
  {
Dsi = sgrid[i]-sgrid[i-1];
Dsip1 = sgrid[i+1]-sgrid[i];
Drk = rgrid[k]-rgrid[k-1];
Drkp1 = rgrid[k+1]-rgrid[k];

sbeta_im1 = -Dsip1/(Dsi*(Dsi+Dsip1));
sbeta_i0 = (Dsip1-Dsi)/(Dsi*Dsip1);
sbeta_ip1 = Dsi/(Dsip1*(Dsi+Dsip1));
rbeta_km1 = -Drkp1/(Drk*(Drk+Drkp1));
rbeta_k0 = (Drkp1-Drk)/(Drk*Drkp1);
rbeta_kp1 = Drk/(Drkp1*(Drk+Drkp1));

msr[stencil(0,0,0)] = sbeta_i0 * rbeta_k0;
//msr[stencil(-1,0,0)] = sbeta_im1 * rbeta_k0;
G_msr += sbeta_im1 * rbeta_k0 * bc_spot_min(strike, divid,
tgrid, Nt, time_index,
sigma_v, alpha_v, beta_v,
sigma_r, alpha_r,
rho_sv, rho_sr, rho_vr,
sgrid, i-1, vgrid, j, rgrid, k,
call_or_put, am);
msr[stencil(1,0,0)] = sbeta_ip1 * rbeta_k0;
msr[stencil(0,0,-1)] = sbeta_i0 * rbeta_km1;
msr[stencil(0,0,1)] = sbeta_i0 * rbeta_kp1;
//msr[stencil(-1,0,-1)] = sbeta_im1 * rbeta_km1;
G_msr += sbeta_im1 * rbeta_km1 * bc_spot_min(strike, divid,
 tgrid, Nt, time_index,
```

```
 sigma_v, alpha_v, beta_v,
 sigma_r, alpha_r,
 rho_sv, rho_sr, rho_vr,
 sgrid, i-1, vgrid, j, rgrid, k-1,
 call_or_put, am);
msr[stencil(1,0,-1)] = sbeta_ip1 * rbeta_km1;
//msr[stencil(-1,0,1)] = sbeta_im1 * rbeta_kp1;
G_msr += sbeta_im1 * rbeta_kp1 * bc_spot_min(strike, divid,
 tgrid, Nt, time_index,
 sigma_v, alpha_v, beta_v,
 sigma_r, alpha_r,
 rho_sv, rho_sr, rho_vr,
 sgrid, i-1, vgrid, j, rgrid, k+1,
 call_or_put, am);
msr[stencil(1,0,1)] = sbeta_ip1 * rbeta_kp1;
  }
  }
  }
  // Mixted VR
  {
// V=Vmin or Vmax or R=Rmin or Rmax -->> Mixted VR = 0
//if (j==0) // mx_vr = 0 by v=0
//if (j==Nv) // mx_vr = 0 by Neumann (Not in the computational domain)
//if (k==0) // mx_vr = 0 by Neumann
//if (k==Nr) // mx_vr = 0 by Neumann
if ((0<j) && (j<Nv-1) && (0<k) && (k<Nr)) // Strict interior 0<j<Nv-1 and 0<k<Nr
  {
Dvj = vgrid[j]-vgrid[j-1];
Dvjp1 = vgrid[j+1]-vgrid[j];
Drk = rgrid[k]-rgrid[k-1];
Drkp1 = rgrid[k+1]-rgrid[k];

vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));
rbeta_km1 = -Drkp1/(Drk*(Drk+Drkp1));
rbeta_k0 = (Drkp1-Drk)/(Drk*Drkp1);
rbeta_kp1 = Drk/(Drkp1*(Drk+Drkp1));

mvr[stencil(0,0,0)] = vbeta_j0 * rbeta_k0;
mvr[stencil(0,-1,0)] = vbeta_jm1 * rbeta_k0;
```

```
mvr[stencil(0,1,0)] = vbeta_jp1 * rbeta_k0;
mvr[stencil(0,0,-1)] = vbeta_j0 * rbeta_km1;
mvr[stencil(0,0,1)] = vbeta_j0 * rbeta_kp1;
mvr[stencil(0,-1,-1)] = vbeta_jm1 * rbeta_km1;
mvr[stencil(0,1,-1)] = vbeta_jp1 * rbeta_km1;
mvr[stencil(0,-1,1)] = vbeta_jm1 * rbeta_kp1;
mvr[stencil(0,1,1)] = vbeta_jp1 * rbeta_kp1;
  }
else // j==0 or j==Nv-1 or k==0 or k==Nr
  {
if ((0<k) && (k<Nr) && (0<j)) // Otherwise mx_vr = 0. So now j==Nv-1 (with 0<k<N
  {
Dvj = vgrid[j]-vgrid[j-1];
Dvjp1 = vgrid[j+1]-vgrid[j];
Drk = rgrid[k]-rgrid[k-1];
Drkp1 = rgrid[k+1]-rgrid[k];

vbeta_jm1 = -Dvjp1/(Dvj*(Dvj+Dvjp1));
vbeta_j0 = (Dvjp1-Dvj)/(Dvj*Dvjp1);
vbeta_jp1 = Dvj/(Dvjp1*(Dvj+Dvjp1));
rbeta_km1 = -Drkp1/(Drk*(Drk+Drkp1));
rbeta_k0 = (Drkp1-Drk)/(Drk*Drkp1);
rbeta_kp1 = Drk/(Drkp1*(Drk+Drkp1));

mvr[stencil(0,0,0)] = vbeta_j0 * rbeta_k0;
mvr[stencil(0,-1,0)] = vbeta_jm1 * rbeta_k0;
//mvr[stencil(0,1,0)] = vbeta_jp1 * rbeta_k0;
G_mvr += vbeta_jp1 * rbeta_k0 * bc_var_max(strike, divid,
   tgrid, Nt, time_index,
   sigma_v, alpha_v, beta_v,
   sigma_r, alpha_r,
   rho_sv, rho_sr, rho_vr,
   sgrid, i, vgrid, j+1, rgrid, k,
   call_or_put, am);
mvr[stencil(0,0,-1)] = vbeta_j0 * rbeta_km1;
mvr[stencil(0,0,1)] = vbeta_j0 * rbeta_kp1;
mvr[stencil(0,-1,-1)] = vbeta_jm1 * rbeta_km1;
//mvr[stencil(0,1,-1)] = vbeta_jp1 * rbeta_km1;
G_mvr += vbeta_jp1 * rbeta_km1 * bc_var_max(strike, divid,
tgrid, Nt, time_index,
sigma_v, alpha_v, beta_v,
```

```
sigma_r, alpha_r,
rho_sv, rho_sr, rho_vr,
sgrid, i, vgrid, j+1, rgrid, k-1,
call_or_put, am);
mvr[stencil(0,-1,1)] = vbeta_jm1 * rbeta_kp1;
//mvr[stencil(0,1,1)] = vbeta_jp1 * rbeta_kp1;
G_mvr += vbeta_jp1 * rbeta_kp1 * bc_var_max(strike, divid,
tgrid, Nt, time_index,
sigma_v, alpha_v, beta_v,
sigma_r, alpha_r,
rho_sv, rho_sr, rho_vr,
sgrid, i, vgrid, j+1, rgrid, k+1,
call_or_put, am);
  }
  }
  }

  // Central point for order_0
  MatrixA0[i][j][k][0] = mixted_sv * msv[0] + mixted_sr * msr[0] + mixted_vr * m
  MatrixA1[i][j][k][0] = order_0 * (SPDE/(SPDE+VPDE+RPDE)) + convection_s * cs[0
  MatrixA2[i][j][k][0] = order_0 * (VPDE/(SPDE+VPDE+RPDE)) + convection_v * cv[0
  MatrixA3[i][j][k][0] = order_0 * (RPDE/(SPDE+VPDE+RPDE)) + convection_r * cr[0

  for (st=1;st<29;st++)
               {
  MatrixA0[i][j][k][st] = mixted_sv * msv[st] + mixted_sr * msr[st] + mixted_vr
  MatrixA1[i][j][k][st] = convection_s * cs[st] + diffusion_s * ds[st];
  MatrixA2[i][j][k][st] = convection_v * cv[st] + diffusion_v * dv[st];
  MatrixA3[i][j][k][st] = convection_r * cr[st] + diffusion_r * dr[st];
  //                    std::cout << "Conv = " << convection_r << " " << diffusi
               }

  //Second members
  G0[i][j][k] = mixted_sv * G_msv * SPDE * VPDE + mixted_sr * G_msr * SPDE * RPD
  G1[i][j][k] = convection_s * G_cs * SPDE + diffusion_s * G_ds * SPDE;
  G2[i][j][k] = convection_v * G_cv * VPDE + diffusion_v * G_dv * VPDE;
  G3[i][j][k] = convection_r * G_cr * RPDE + diffusion_r * G_dr * RPDE;
           }
      }
    }
  /*
```

```
    std::cout << "A0 = " << std::endl;
    print_matrix_k_fixed(Ns,Nv,MatrixA0,1);
    std::cout << "A1 = " << std::endl;
    print_matrix_k_fixed(Ns,Nv,MatrixA1,1);
    std::cout << "A2 = " << std::endl;
    print_matrix_k_fixed(Ns,Nv,MatrixA2,1);

    std::cout << "A3 = " << std::endl;
    print_matrix_k_fixed(Ns,Nv,MatrixA3,1);
    // */

  free(mvr);
  free(msr);
  free(msv);
  free(dr);
  free(cr);
  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 ****M
                                               double ***G0nm1, double ***G1nm1,
                                               double ***Unm1, double ***Y0)
{

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

  for (i=1; i<Ns+1; i++)
    {
  for (j=0; j<Nv; j++)
      {
  for (k=0; k<Nr+1; k++)
          {
  val = Unm1[i][j][k];
  for (st=0; st<29; st++)
```

```
                {
  if (it_exists_stencil(i, Ns, j, Nv, k, Nr, st))
                    {
  // If the point exists.
  point_of_stencil(i, j, k, st, &istencil, &jstencil, &kstencil);
  val += coeff * MatrixA0[i][j][k][st] * Unm1[istencil][jstencil][kstencil];
  val += coeff * MatrixA1[i][j][k][st] * Unm1[istencil][jstencil][kstencil];
  val += coeff * MatrixA2[i][j][k][st] * Unm1[istencil][jstencil][kstencil];
  val += coeff * MatrixA3[i][j][k][st] * Unm1[istencil][jstencil][kstencil];
                    }
                }
  val += coeff * G0nm1[i][j][k];
  val += coeff * G1nm1[i][j][k];
  val += coeff * G2nm1[i][j][k];
  val += coeff * G3nm1[i][j][k];
  Y0[i][j][k] = val;
            }
        }
    }
}

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

  val=0.;

  for (j=0; j<Nv; j++)
    {
  for (k=0; k<Nr+1; k++)
      {
  for (i=1; i<Ns+1; i++)
            {
  val = Y0[i][j][k];
  for (st=0; st<29; st++)
```

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

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

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

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

  for (j=0; j<Nv; j++)
    {
  for (k=0; k<Nr+1; k++)
        {
  // Build the matrix
```

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

  pnl_tridiag_mat_syslin(Sortie, Working_Matrix, Entree);

  for (i=0; i<Ns; i++)
            {
  lhs[i+1][j][k] = 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 ***G1
                                                   double ***Unm1, double ***Y1,
{
  int i, j, k, st;
  double val;
  int istencil, jstencil, kstencil;

  val =0.;

  for (i=1; i<Ns+1; i++)
    {
  for (k=0; k<Nr+1; k++)
        {
  for (j=0; j<Nv; j++)
            {
  val = Y1[i][j][k];
  for (st=0; st<29; st++)
                {
  if (it_exists_stencil(i, Ns, j, Nv, k, Nr, st))
                    {
  // If the point exists.
  point_of_stencil(i, j, k, st, &istencil, &jstencil, &kstencil);
  val += -coeff * MatrixA2[i][j][k][st] * Unm1[istencil][jstencil][kstencil];
                    }
                }
  val += -coeff * G2nm1[i][j][k];
  Sortie[i][j][k] = val;
            }
        }
    }
}

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

```
int i, j, k;

// Only points from i=1 to i=Ns.
// Only points from j=0 to j=Nv-1.
// Only points from k=0 to k=Nr.
// Pentadiagonal matrix.
PnlBandMat *Working_Matrix;
PnlVect *Entree;
PnlVect *Sortie;

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

for (i=1; i<Ns+1; i++)
  {
for (k=0; k<Nr+1; k++)
      {
Working_Matrix = pnl_band_mat_create(Nv,Nv,2,2);
// Build the matrix

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

```
  pnl_band_mat_set (Working_Matrix, Nv-2, Nv-2+1, MatrixA2[i][Nv-2][k][5]);
  //pnl_band_mat_set (Working_Matrix, Nv-2, Nv-2+2, MatrixA2[i][Nv-2][k][6]);
  pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1-2, MatrixA2[i][Nv-1][k][3]);
  pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1-1, MatrixA2[i][Nv-1][k][4]);
  pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1+0, MatrixA2[i][Nv-1][k][0]);
  //pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1+1, MatrixA2[i][Nv-1][k][5]);
  //pnl_band_mat_set (Working_Matrix, Nv-1, Nv-1+2, MatrixA2[i][Nv-1][k][6]);

  // Multiplication by -coeff.
  pnl_band_mat_mult_double(Working_Matrix, -coeff);
  // Sum with the identity matrix.
  for (j=0; j<Nv; 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; j++)
            {
  pnl_vect_set (Entree, j, rhs[i][j][k] + coeff * G2[i][j][k]);
            }

  pnl_band_mat_syslin(Sortie, Working_Matrix, Entree);

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

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

static void computation_explicit_syslin_rate_matrix(double coeff,
                                                    double *sgrid, int Ns, doubl
                                                    double ****MatrixA0, double
                                                    double ***G0nm1, double ***G
                                                    double ***Unm1, double ***Y2
```

```
{
  int i, j, k, st;
  double val;
  int istencil, jstencil, kstencil;

  val=0.;

  for (i=1; i<Ns+1; i++)
    {
  for (j=0; j<Nv; j++)
        {
  for (k=0; k<Nr+1; k++)
            {
  val = Y2[i][j][k];
  for (st=0; st<29; st++)
                {
  if (it_exists_stencil(i, Ns, j, Nv, k, Nr, st))
                    {
  // If the point exists.
  point_of_stencil(i, j, k, st, &istencil, &jstencil, &kstencil);
  val += -coeff * MatrixA3[i][j][k][st] * Unm1[istencil][jstencil][kstencil];
                    }
                }
  val += -coeff * G3nm1[i][j][k];
  Sortie[i][j][k] = val;
            }
        }
    }
}

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

  // Only points from i=1 to i=Ns.
  // Only points from j=0 to j=Nv-1.
  // Only points from k=0 to k=Nr.
```

```
PnlTridiagMat *Identity_Matrix;
PnlTridiagMat *Working_Matrix;
PnlVect *Entree;
PnlVect *Sortie;

Identity_Matrix = pnl_tridiag_mat_create_from_two_double(Nr+1, 1.0, 0.0); // I
Working_Matrix = pnl_tridiag_mat_create(Nr+1);
Entree = pnl_vect_create(Nr+1);
Sortie = pnl_vect_create(Nr+1);

for (i=1; i<Ns+1; i++)
  {
for (j=0; j<Nv; j++)
      {
// Build the matrix

//pnl_tridiag_mat_set (Working_Matrix, 0, -1, MatrixA3[i][j][0][11]);
pnl_tridiag_mat_set (Working_Matrix, 0,  0, MatrixA3[i][j][0][0]);
pnl_tridiag_mat_set (Working_Matrix, 0,  1, MatrixA3[i][j][0][20]);
for (k=1; k<Nr; k++)
          {
pnl_tridiag_mat_set (Working_Matrix, k, -1, MatrixA3[i][j][k][11]);
pnl_tridiag_mat_set (Working_Matrix, k,  0, MatrixA3[i][j][k][0]);
pnl_tridiag_mat_set (Working_Matrix, k,  1, MatrixA3[i][j][k][20]);
          }
pnl_tridiag_mat_set (Working_Matrix, Nr, -1, MatrixA3[i][j][Nr][11]);
pnl_tridiag_mat_set (Working_Matrix, Nr,  0, MatrixA3[i][j][Nr][0]);
//pnl_tridiag_mat_set (Working_Matrix, Nr,  1, MatrixA3[i][j][Nr][20]);


// 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);

//            pnl_tridiag_mat_print(Working_Matrix);

// Build the vectors.
for (k=0; k<Nr+1; k++)
          {
pnl_vect_set (Entree, k, rhs[i][j][k] + coeff * G3[i][j][k]);
```

```
            }

  //std::cout << "Working matrix" << std::endl;
  //pnl_tridiag_mat_print(Working_Matrix);

  pnl_tridiag_mat_syslin(Sortie, Working_Matrix, Entree);

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

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


static int ADI_HHW
(int flat_flag,double S0, double V0, double R0,
 double strike, double divid, double t,
 double sigma_r, double alpha_r,
 double sigma_v, double alpha_v, double beta_v,
 double rho_sv, double rho_sr, double rho_vr,
 double Smax, double Vmax, double Rmax,
 double Coeff_s, double Coeff_v, double Coeff_r,
 double Sleft, double Sright, double Center_r,
 int Ns, int Nv, int Nr, int Nt,
 double theta,  int scheme, int call_or_put, int am,
 double *ptprice, double *ptdelta)
{

  int TimeIndex, i, j, k, st;
  int IndexS, IndexV, IndexR;

  double deltat;

  double price_sd_vd_rd, price_sd_vu_rd, price_su_vd_rd, price_su_vu_rd;
```

```
double price_sd_vd_ru, price_sd_vu_ru, price_su_vd_ru, price_su_vu_ru;

// 1-vectors
double *sgrid;
double *vgrid;
double *rgrid;
double *tgrid;

// 3-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 (=4-vectors)
double ****MatrixA0nm1;
double ****MatrixA1nm1;
double ****MatrixA2nm1;
double ****MatrixA3nm1;
//double ****MatrixA0n;
double ****MatrixA1n;
double ****MatrixA2n;
double ****MatrixA3n;

// Memory allocation of 1-vectors.
sgrid=(double *)malloc((Ns+1)*sizeof(double));
vgrid=(double *)malloc((Nv+1)*sizeof(double));
rgrid=(double *)malloc((Nr+1)*sizeof(double));
tgrid=(double *)malloc((Nt+1)*sizeof(double));
grid_generation_HHW_spot(sgrid, Sleft, Sright, Smax, Ns, Coeff_s);
grid_generation_HHW_variance(vgrid, Vmax, Nv, Coeff_v, V0);
```

```
  grid_generation_HHW_rate(rgrid, Rmax, Nr, Coeff_r, Center_r);
  for (i=0; i<=Nt; i++)
tgrid[i] = t_vect[i];


  // Memory allocation of 3-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**));
  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*));
  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*));
  for (j = 0; j < Nv+1; j++)
        {
  Unm1[i][j] = (double*) malloc((Nr+1) * sizeof(double));
  Utmp[i][j] = (double*) malloc((Nr+1) * sizeof(double));
  Y0[i][j] = (double*) malloc((Nr+1) * sizeof(double));
  Y1[i][j] = (double*) malloc((Nr+1) * sizeof(double));
  Y2[i][j] = (double*) malloc((Nr+1) * sizeof(double));
```

```
G0nm1[i][j] = (double*) malloc((Nr+1) * sizeof(double));
G1nm1[i][j] = (double*) malloc((Nr+1) * sizeof(double));
G2nm1[i][j] = (double*) malloc((Nr+1) * sizeof(double));
G3nm1[i][j] = (double*) malloc((Nr+1) * sizeof(double));
//G0n[i][j] = (double*) malloc((Nr+1) * sizeof(double));
G1n[i][j] = (double*) malloc((Nr+1) * sizeof(double));
G2n[i][j] = (double*) malloc((Nr+1) * sizeof(double));
G3n[i][j] = (double*) malloc((Nr+1) * sizeof(double));
      }
  }

// Memory allocation of matrix (=4-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((Nr+1) * sizeof(double*));
MatrixA1nm1[i][j] = (double**) malloc((Nr+1) * sizeof(double*));
MatrixA2nm1[i][j] = (double**) malloc((Nr+1) * sizeof(double*));
MatrixA3nm1[i][j] = (double**) malloc((Nr+1) * sizeof(double*));
//MatrixA0n[i][j] = (double**) malloc((Nr+1) * sizeof(double*));
MatrixA1n[i][j] = (double**) malloc((Nr+1) * sizeof(double*));
MatrixA2n[i][j] = (double**) malloc((Nr+1) * sizeof(double*));
MatrixA3n[i][j] = (double**) malloc((Nr+1) * sizeof(double*));
for (k=0; k<Nr+1; k++)
```

```
          {
MatrixA0nm1[i][j][k] = (double*) malloc(29 * sizeof(double));
MatrixA1nm1[i][j][k] = (double*) malloc(29 * sizeof(double));
MatrixA2nm1[i][j][k] = (double*) malloc(29 * sizeof(double));
MatrixA3nm1[i][j][k] = (double*) malloc(29 * sizeof(double));
//MatrixA0n[i][j][k] = (double*) malloc(29 * sizeof(double));
MatrixA1n[i][j][k] = (double*) malloc(29 * sizeof(double));
MatrixA2n[i][j][k] = (double*) malloc(29 * sizeof(double));
MatrixA3n[i][j][k] = (double*) malloc(29 * sizeof(double));
          }
      }
  }

// Initialization
for (i=0; i<Ns+1; i++)
  {
for (j=0; j<Nv+1; j++)
      {
for (k=0; k<Nr+1; k++)
          {
if (call_or_put!=0)
              {
// TERMINAL CONDITION FOR CALL AND PUT
Unm1[i][j][k] = MAX(call_or_put*(sgrid[i] - strike),0.0);
Utmp[i][j][k] = MAX(call_or_put*(sgrid[i] - strike),0.0);
              }
else // call_or_put == 0
              {
// TERMINAL CONDITION FOR VASICEK ZERO COUPON BOND
Unm1[i][j][k] = 1.;
Utmp[i][j][k] = 1.;
              }


Y0[i][j][k] = 0.;
Y1[i][j][k] = 0.;
Y2[i][j][k] = 0.;
//Y3[i][j][k] = 0.;
G0nm1[i][j][k] = 0.;
G1nm1[i][j][k] = 0.;
G2nm1[i][j][k] = 0.;
```

```
G3nm1[i][j][k] = 0.;
//G0n[i][j][k] = 0.;
G1n[i][j][k] = 0.;
G2n[i][j][k] = 0.;
G3n[i][j][k] = 0.;
          }
      }
  }

for (i=0; i<Ns+1; i++)
  {
for (j=0; j<Nv+1; j++)
      {
for (k=0; k<Nr+1; k++)
          {
for (st=0; st<29; st++)
              {
MatrixA0nm1[i][j][k][st] = 0.;
MatrixA1nm1[i][j][k][st] = 0.;
MatrixA2nm1[i][j][k][st] = 0.;
MatrixA3nm1[i][j][k][st] = 0.;
//MatrixA0n[i][j][k][st] = 0.;
MatrixA1n[i][j][k][st] = 0.;
MatrixA2n[i][j][k][st] = 0.;
MatrixA3n[i][j][k][st] = 0.;
              }
          }
      }
  }
```

```
// Time Step
deltat=t/((double)Nt);

// Finite Difference Cycle.
for (TimeIndex=1;TimeIndex<Nt;TimeIndex++)
```

```
  {

if (am==1)
      {
// American condition = Unknowns >= payoff
for (i=0; i<Ns+1; i++)
          {
for (j=0; j<Nv+1; j++)
              {
for (k=0; k<Nr+1; k++)
                  {
if (call_or_put==1) //Call
                      {
Unm1[i][j][k] = MAX(Unm1[i][j][k],MAX(sgrid[i] - strike,0.0));
                      }
if (call_or_put==-1) //Put
                      {
Unm1[i][j][k] = MAX(Unm1[i][j][k],MAX(strike - sgrid[i],0.0));
                      }
                  }
              }
          }
      }



if (scheme==0) // Douglas scheme
      {
// Compute the elements at time step n (except A0n and G0n which are not used,
// so they are erased by the next call to build matrix).
build_all_matrix(flat_flag,strike, divid, tgrid, Nt, TimeIndex, sigma_v, alpha
 rho_sv, rho_sr, rho_vr, sgrid, Ns, vgrid, Nv, rgrid, Nr, call_or_put, am,
 MatrixA0nm1, MatrixA1n, MatrixA2n, MatrixA3n,
 G0nm1, G1n, G2n, G3n);


// Compute the elements at time step n-1.
build_all_matrix(flat_flag,strike, divid, tgrid, Nt, TimeIndex-1 , sigma_v, al
 rho_sv, rho_sr, rho_vr, sgrid, Ns, vgrid, Nv, rgrid, Nr, call_or_put, am,
 MatrixA0nm1, MatrixA1nm1, MatrixA2nm1, MatrixA3nm1,
 G0nm1, G1nm1, G2nm1, G3nm1);
```

```
 // 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,
sgrid, Ns, vgrid, Nv, rgrid, Nr,
MatrixA0nm1, MatrixA1nm1, MatrixA2nm1, MatrixA3nm1,
G0nm1, G1nm1, G2nm1, G3nm1,
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,rgrid, Nr,
 MatrixA0nm1, MatrixA1nm1, MatrixA2nm1, MatrixA3nm1,
 G0nm1, G1nm1, G2nm1, G3nm1,
 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, rgrid, Nr,
 MatrixA0nm1, MatrixA1n, MatrixA2n, MatrixA3n,
 G0nm1, G1n, G2n, G3n,
 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, rgrid, Nr,
MatrixA0nm1, MatrixA1nm1, MatrixA2nm1, MatrixA3nm1,
G0nm1, G1nm1, G2nm1, G3nm1,
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, rgrid, Nr,
MatrixA0nm1, MatrixA1n, MatrixA2n, MatrixA3n,
G0nm1, G1n, G2n, G3n,
Utmp, Y2);
 // Utmp = Y2 - theta*Dt A3[n-1] * U[n-1] - theta*Dt G3[n-1]
 computation_explicit_syslin_rate_matrix(theta * deltat,
 sgrid, Ns, vgrid, Nv, rgrid, Nr,
 MatrixA0nm1, MatrixA1nm1, MatrixA2nm1, MatrixA3nm1,
 G0nm1, G1nm1, G2nm1, G3nm1,
 Unm1, Y2, Utmp);

 // Y3 = (Id - theta*Dt A3[n])^-1 ( Utmp + theta*Dt G3[n] )
```

```
computation_implicit_syslin_rate_matrix(theta * deltat,
sgrid, Ns, vgrid, Nv, rgrid, Nr,
MatrixA0nm1, MatrixA1n, MatrixA2n, MatrixA3n,
G0nm1, G1n, G2n, G3n,
Utmp, Unm1); // /!\  with Y3=Un


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

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

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

      }
  }
```

```
// 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,S0);
// Find the index in vgrid corresponding to the variance V0 of the asset.
IndexV = lower_index(vgrid,Nv+1,V0);
// Find the index in rgrid corresponding to the rate R0 of the asset.
IndexR = lower_index(rgrid,Nr+1,R0);

//    std::cout << "IndexS=" << IndexS << " IndexV=" << IndexV << " IndexR=" <
//    std::cout << "S ~ " << sgrid[IndexS] << " V ~ " << vgrid[IndexV] << " R
//    std::cout << "Value=" << Unm1[IndexS][IndexV][IndexR] << std::endl;

// Compute first the delta (using *ptprice as temporary variable). We do an in
price_sd_vd_rd = Unm1[IndexS][IndexV][IndexR];
```

```
  price_sd_vu_rd = Unm1[IndexS][IndexV+1][IndexR];
  //    std::cout << price_sd_vd_rd << " " << price_sd_vu_rd << std::endl;
  price_su_vd_rd = Unm1[IndexS+1][IndexV][IndexR];
  price_su_vu_rd = Unm1[IndexS+1][IndexV+1][IndexR];
  //    std::cout << price_su_vd_rd << " " << price_su_vu_rd << std::endl;
  price_sd_vd_ru = Unm1[IndexS][IndexV][IndexR+1];
  price_sd_vu_ru = Unm1[IndexS][IndexV+1][IndexR+1];
  //    std::cout << price_sd_vd_ru << " " << price_sd_vu_ru << std::endl;
  price_su_vd_ru = Unm1[IndexS+1][IndexV][IndexR+1];
  price_su_vu_ru = Unm1[IndexS+1][IndexV+1][IndexR+1];
  //    std::cout << price_su_vd_ru << " " << price_su_vu_ru << std::endl;

  *ptprice=triple_interpolation(
price_sd_vd_rd, price_sd_vu_rd, price_su_vd_rd, price_su_vu_rd,
price_sd_vd_ru, price_sd_vu_ru, price_su_vd_ru, price_su_vu_ru,
sgrid[IndexS], sgrid[IndexS+1],
vgrid[IndexV], vgrid[IndexV+1],
rgrid[IndexR], rgrid[IndexR+1],
sgrid[IndexS], V0, R0);

  *ptdelta =
    (triple_interpolation(
                          price_sd_vd_rd, price_sd_vu_rd, price_su_vd_rd, price_
                          price_sd_vd_ru, price_sd_vu_ru, price_su_vd_ru, price_
                          sgrid[IndexS], sgrid[IndexS+1],
                          vgrid[IndexV], vgrid[IndexV+1],
                          rgrid[IndexR], rgrid[IndexR+1],
                          sgrid[IndexS+1], V0, R0)
     - *ptprice)
    /(sgrid[IndexS+1] - sgrid[IndexS]);

  // Next compute the price. We do an interpolation.
  *ptprice=triple_interpolation(
price_sd_vd_rd, price_sd_vu_rd, price_su_vd_rd, price_su_vu_rd,
price_sd_vd_ru, price_sd_vu_ru, price_su_vd_ru, price_su_vu_ru,
sgrid[IndexS], sgrid[IndexS+1],
vgrid[IndexV], vgrid[IndexV+1],
rgrid[IndexR], rgrid[IndexR+1],
S0, V0, R0);

  // Memory desallocation of matrix (=4-vectors)
```

```
for (i=0; i<Ns+1; i++)
  {
for (j=0; j<Nv+1; j++)
      {
for (k=0; k<Nr+1; k++)
          {
free(MatrixA0nm1[i][j][k]);
free(MatrixA1nm1[i][j][k]);
free(MatrixA2nm1[i][j][k]);
free(MatrixA3nm1[i][j][k]);
//free(MatrixA0n[i][j][k]);
free(MatrixA1n[i][j][k]);
free(MatrixA2n[i][j][k]);
free(MatrixA3n[i][j][k]);
          }
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++)
  {
for (j=0; j<Nv+1; j++)
      {
free(Unm1[i][j]);
free(Utmp[i][j]);
free(Y0[i][j]);
free(Y1[i][j]);
free(Y2[i][j]);
//free(Y3[i][j])
free(G0nm1[i][j]);
free(G1nm1[i][j]);
free(G2nm1[i][j]);
free(G3nm1[i][j]);
//free(G0n[i][j]);
free(G1n[i][j]);
free(G2n[i][j]);
free(G3n[i][j]);
      }
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);

  // Memory desallocation of 1-vectors
  free(sgrid);
  free(vgrid);
  free(rgrid);
  free(tgrid);

  return 0;
}

/*Compute Price Option*/
int  Fd_Hout_HesHw(int am, double S0, NumFunc_1  *p, double t, double divid, int
{
  double strike;
  int call_or_put;
  double Smax,Vmax,Rmax,Sleft,Sright,Coeff_s,Coeff_v,Coeff_r,Center_r;
  int NinterpolZCB;
  double theta;
  int scheme;
  int i;
  int n_price;
  double *t_interpol;
  double *forward;
  double *derive_forward;

  R0_flat=R0;
  theta=0.5;//Theta schema
  scheme=0.;//Douglas Scheme

  NinterpolZCB = Nt*1000;

   strike = p->Par[0].Val.V_DOUBLE;
```

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

  Smax = 5*strike;
  Vmax = 10.*V0;
  Rmax = 0.5;
  Sleft = MAX(1/2, exp(-t/4))*strike;
  Sright = strike;
  Coeff_s = strike/20; // Environ entre 2 et 5 ou fraction de Ns/2
  Coeff_v = Vmax/500;
  Coeff_r = Rmax/400;

  // R-grid centered around Center_r = R0 or Mean-reversion
  Center_r=R0;

  /*
.European Call
.                   Vol max => Dirichlet = s*exp(-divid*t)
.                   ------------------------
.Spot 0             |                       | Spot max => Neumann
.=                  |                       | = exp(-divid*t)
.0                  |                       |
.                   |                       |
.                   |_______________________|
.                   Outflow boundary
  */
  /*
.European Put
.                   Vol max => Dirichlet = strike*exp(-r*t)
.                   ------------------------
.Spot 0 |                     | Spot max => Neumann
.=                  |                       | = 0
.strike*exp(-r*t)   |                       |
.                   |                       |
.                   |_______________________|
.                   Outflow boundary
  */

  /*ZCB*/
```

```
  // Read the values of pm and tm
  init_tr = curve;
  n_price = lecture_tr();
  /* We search in initialyield.dat the biggest value before time T */
  if (t > tm[n_price - 1])
    {
  printf("\ nError : time bigger than the last time value entered in initialyiel
    }

  // Compute interpolation on a very fine grid given pm, tm and n_price. Return
  NinterpolZCB = tm[n_price - 1] * NinterpolZCB;
  initial_yield = (double *)malloc((NinterpolZCB + 1) * sizeof(double));
  t_interpol = (double *)malloc((NinterpolZCB + 1) * sizeof(double));
  for (i = 0; i <= NinterpolZCB; i++)
t_interpol[i] = i * tm[n_price - 1]/NinterpolZCB;
   for (i = 0; i <= NinterpolZCB; i++)
{
  initial_yield[i] = 0.;
}
  interpole(n_price, NinterpolZCB, t_interpol);

  forward = (double *)malloc((NinterpolZCB + 1) * sizeof(double));
  derive_forward = (double *)malloc((NinterpolZCB + 1) * sizeof(double));

  // Compute the forward rate and the derivative of the forward rate given a ver

  for (i = 0; i <= NinterpolZCB; i++)
{
  forward[i] = 0.;
  derive_forward[i] = 0.;
}

  compute_forward(initial_yield, t_interpol, NinterpolZCB, forward, derive_forwa
  // We obtain the forward and derivative of forward rate on a very fine grid.

  initial_forward = (double *)malloc((Nt + 1) * sizeof(double));
  initial_derive_forward = (double *)malloc((Nt + 1) * sizeof(double));
  t_vect = (double *)malloc((Nt + 2) * sizeof(double));

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

```
  initial_forward[i] = 0.;
  initial_derive_forward[i] = 0.;
}

  for (i = 0; i <= Nt+1; i++)
t_vect[i] = i * t/(double)Nt;


  extract_forward(forward, derive_forward, t_interpol, NinterpolZCB, t_vect,Nt);

  ADI_HHW(flat_flag,S0, V0, R0, strike, divid, t,
  sigma_r, alpha_r, sigma_v, alpha_v, beta_v, rho_sv, rho_sr, rho_vr,
  Smax, Vmax, Rmax, Coeff_s, Coeff_v, Coeff_r, Sleft, Sright, Center_r, Ns, Nv,
  theta,scheme, call_or_put, am,
  ptprice,  ptdelta);

  // Free all the temporary arrays
    free(Pm);
    free(tm);
    free(t_vect);
    free(initial_derive_forward);
    free(initial_forward);
    free(initial_yield);
free(forward);
free(derive_forward);
free(t_interpol);

  return OK;
}

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

  divid = ptMod->divid.Val.V_DOUBLE;
  return Fd_Hout_HesHw(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,divid, ptMod->flat_flag.Val.
   MOD(GetYield)(ptMod),
```

```
   MOD(GetCurve)(ptMod),
   ptMod->kr.Val.V_PDOUBLE,
   ptMod->Sigmar.Val.V_PDOUBLE,
   ptMod->V0.Val.V_PDOUBLE
   ,ptMod->kV.Val.V_PDOUBLE,
   ptMod->thetaV.Val.V_PDOUBLE,
   ptMod->SigmaV.Val.V_PDOUBLE,
   ptMod->RhoSr.Val.V_PDOUBLE,
   ptMod->RhoSV.Val.V_PDOUBLE,
   ptMod->RhorV.Val.V_PDOUBLE,
   Met->Par[0].Val.V_PINT,
   Met->Par[1].Val.V_PINT,
   Met->Par[2].Val.V_PINT,
   Met->Par[3].Val.V_PINT,
   &(Met->Res[0].Val.V_DOUBLE),&(Met->Res[1].Val.V_DOUBLE));
}

static int CHK_OPT(FD_Hout)(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)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->HelpFilenameHint = "fd_hout_heshw";
      Met->Par[0].Val.V_INT = 100;
  Met->Par[1].Val.V_INT = 50;
      Met->Par[2].Val.V_INT = 10;
  Met->Par[3].Val.V_INT = 10;
    }

  return OK;
}

PricingMethod MET(FD_Hout) =
```

```
{
  "FD_Hout",
  { {"N steps time", INT, {100}, ALLOW},
{"SpaceStepNumber S", INT2, {100}, ALLOW}, {"SpaceStepNumber V", INT2, {100}, AL
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_Hout),
  { {"Price", DOUBLE, {100}, FORBID},{"Delta", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(FD_Hout),
  CHK_ok,
  MET(Init)
};
```