

[Help](#)

```
#include "
href../../../../mod/uvm1d/uvm1d_pad/uvm1d_pad_h_src.pdfuvm1d_pad.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_tridiag_matrix.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2012+2) //The "#els
static int CHK_OPT(FD_UVM)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(FD_UVM)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

static void grid_construction(double *ptsgrid, double sstar, double **ptptgrid,
{
    int i, j;
    for (j = 0; j < sizeP; j++)
    {
        for (i = 0; i < sizeS; i++)
        {
            ptptgrid[j][i] = ptsgrid[i] * ptsgrid[j] / sstar;
        }
    }
}

static int nearest(double *ptgrid, int size, double value, int begin)
{
    double value_nearest = ABS(ptgrid[begin] - value);
    double index_nearest = -1;
    int i;
    for (i = begin; i < size; i++)
    {
        if (ABS(ptgrid[i] - value) <= value_nearest)
        {
            value_nearest = ABS(ptgrid[i] - value);
            index_nearest = i;
        }
    }
}
```

```

    }
}
return index_nearest;
}

```

```

static int nearest_lower(double *ptgrid, int size, double value)
{
    int i = 0;
    while ((i < size) && (ptgrid[i] <= value))
    {
        i++;
    }
    i--;
    return i;
}

```

```

static double interpolation_linear(double x, double y, double fx, double fy, double z)
{
    if (x != y)
    {
        return fx + (fy - fx) * (z - x) / (y - x);
    }
    else
    {
        return (fx + fy) / 2.; // Just in case... Or return fx;
    }
}

```

```

double interpolation_xy(double slow, double shigh, double plow, double phigh,
                      double V_slow_plow, double V_shigh_plow,
                      double V_slow_phigh, double V_shigh_phigh,
                      double s, double p)
{
    double V_ll_hl = interpolation_linear(slow, shigh, V_slow_plow, V_shigh_plow,
    double V_lh_hh = interpolation_linear(slow, shigh, V_slow_phigh, V_shigh_phigh,
    return interpolation_linear(plow, phigh, V_ll_hl, V_lh_hh, p);
}

```

```

double interpolation_diagonal(double slow, double shigh, double plow, double phigh,
                           double V_slow_plow, double V_shigh_plow,

```

```

double V_slow_phigh, double V_shigh_phigh,
double s, double p)
{
    if (ABS(s - p) < ABS(s * 0.00001))
    {
        return interpolation_linear(plow, phigh, V_slow_plow, V_shigh_phigh, p);
    }
    else
    {
        // return NAN; // Interpolation, only on the diagonal.
        return 0.;
    }
}

static void resolution_pde_sigma_constant(double *Space_points, int size_Space_p
double T, int NT,
double r, double divid, double sigma,
double theta, double omega,
double *Sortie)
{
    int i, j;
    double coeff1, coeff2, coeff3, DeltaSpaceUp, DeltaSpaceDown, DeltaSpaceDouble;
    double tau;
    double *upper_d_lhs, *diagonal_lhs, *lower_d_lhs;
    double *upper_d_rhs, *diagonal_rhs, *lower_d_rhs;
    PnlVect *Solution;
    PnlVect *Vector;
    PnlTridiagMat *Mat_lhs;
    PnlTridiagMat *Mat_rhs;
    PnlTridiagMat *Mat_for_non_const_syslin;

    /*Initialization*/
    tau = T / NT;

    /*Memory Allocation*/
    upper_d_lhs = (double *)malloc(size_Space_points * sizeof(double));
    diagonal_lhs = (double *)malloc(size_Space_points * sizeof(double));
    lower_d_lhs = (double *)malloc(size_Space_points * sizeof(double));
    upper_d_rhs = (double *)malloc(size_Space_points * sizeof(double));
    diagonal_rhs = (double *)malloc(size_Space_points * sizeof(double));

```

```

lower_d_rhs = (double *)malloc(size_Space_points * sizeof(double));

/*Left hand side*/
/*
V^n+1_i
- tau*theta*sigma^2 S_i^2/2 *
( (V^n+1_i+1 - V^n+1_i)/(S_i+1-S_i)
  -(V^n+1_i - V^n+1_i-1)/(S_i-S_i-1)) / ((S_i+1 - S_i-1)/2)
- tau*theta*r*S_i * ( (V^n+1_i+1 - V^n+1_i)/(S_i+1-S_i) *omega/2
                      + (V^n+1_i - V^n+1_i-1)/(S_i-S_i-1) *(2-omega)/2)
+r*tau*theta *V^n+1_i
*/
for (i = 1; i < size_Space_points - 1; i++)
{
    DeltaSpaceDown = 1.0 / (Space_points[i] - Space_points[i - 1]);
    DeltaSpaceUp = 1.0 / (Space_points[i + 1] - Space_points[i]);
    DeltaSpaceDouble = 2.0 / (Space_points[i + 1] - Space_points[i - 1]);
    coeff1 = sigma * Space_points[i];
    coeff1 = tau * theta * coeff1 * coeff1 / 2.0;
    coeff2 = tau * theta * r * Space_points[i];
    diagonal_lhs[i] = 1.0
                      - coeff1 * (-1.0) * (DeltaSpaceUp + DeltaSpaceDown) * De
                      - coeff2 * (-omega / 2.0 * DeltaSpaceUp + (2.0 - omega)
                      + r * tau * theta;
    upper_d_lhs[i] = -coeff1 * DeltaSpaceUp * DeltaSpaceDouble
                    - coeff2 * omega / 2.0 * DeltaSpaceUp;
    // Be careful of the index for lower band.
    lower_d_lhs[i - 1] = -coeff1 * DeltaSpaceDown * DeltaSpaceDouble
                        + coeff2 * (2.0 - omega) / 2.0 * DeltaSpaceDown;
}

/*Right hand side*/
/*
V^n_i
+ tau*(1-theta)*sigma^2 S_i^2/2 *
( (V^n_i+1 - V^n_i)/(S_i+1-S_i)
  -(V^n_i - V^n_i-1)/(S_i-S_i-1)) / ((S_i+1 - S_i-1)/2)
+ tau*(1-theta)*r*S_i * ( (V^n_i+1 - V^n_i)/(S_i+1-S_i) *omega/2
                        + (V^n_i - V^n_i-1)/(S_i-S_i-1) *(2-omega)/2)
-r*tau*(1-theta) *V^n_i

```

```

*/

for (i = 1; i < size_Space_points - 1; i++)
{
    DeltaSpaceDown = 1.0 / (Space_points[i] - Space_points[i - 1]);
    DeltaSpaceUp = 1.0 / (Space_points[i + 1] - Space_points[i]);
    DeltaSpaceDouble = 2.0 / (Space_points[i + 1] - Space_points[i - 1]);
    coeff1 = sigma * Space_points[i];
    coeff1 = tau * theta * coeff1 * coeff1 / 2.0;
    coeff2 = tau * theta * r * Space_points[i];
    diagonal_rhs[i] = 1.0
                    + coeff1 * (-DeltaSpaceUp - DeltaSpaceDown) * DeltaSpace
                    + coeff2 * (-omega / 2.0 * DeltaSpaceUp + (2.0 - omega)
                    - r * tau * (1.0 - theta);
    upper_d_rhs[i] = coeff1 * DeltaSpaceUp * DeltaSpaceDouble
                    + coeff2 * omega / 2.0 * DeltaSpaceUp;
    // Be careful of the index for lower band.
    lower_d_rhs[i - 1] = coeff1 * DeltaSpaceDown * DeltaSpaceDouble
                    - coeff2 * (2.0 - omega) / 2.0 * DeltaSpaceDown;
}

/* Coefficients consistant with boundary conditions */

/*
Si i=imin+1 alors

$$(V^{n_i} - V^{n_{i-1}})/(S_i - S_{i-1}) = (V^{n_{i+1}} - V^{n_i})/(S_{i+1} - S_i)$$

donc
//Left hand side

$$V^{n+1}_i - \tau * \theta * r * S_i * (V^{n+1}_{i+1} - V^{n+1}_i)/(S_{i+1} - S_i) + r * \tau * \theta * V^{n+1}_i$$

//Right hand side

$$V^n_i + \tau * (1 - \theta) * r * S_i * (V^{n_{i+1}} - V^n_i)/(S_{i+1} - S_i) - r * \tau * (1 - \theta) * V^n_i$$

*/

coeff3 = r * tau * Space_points[0];
DeltaSpaceUp = 1.0 / (Space_points[1] - Space_points[0]);

diagonal_lhs[0] = 1.0 + theta * coeff3 * DeltaSpaceUp + r * tau * theta;

```

```

upper_d_lhs[0] = -theta * coeff3 * DeltaSpaceUp;
diagonal_rhs[0] = 1.0 - (1.0 - theta) * coeff3 * DeltaSpaceUp - r * tau * (1.0
upper_d_rhs[0] = (1.0 - theta) * coeff3 * DeltaSpaceUp;

/*
Si i=imax-1 alors
(V^n_i - V^n_{i-1})/(S_i-S_{i-1}) = (V^n_{i+1} - V^n_i)/(S_{i+1}-S_i)
donc
//Left hand side
V^{n+1}_i
- tau*theta*r*S_i * (V^{n+1}_i - V^{n+1}_{i-1})/(S_i-S_{i-1})
+r*tau*theta * V^{n+1}_i
//Right hand side
V^n_i
+ tau*(1-theta)*r*S_i * (V^n_i - V^n_{i-1})/(S_i-S_{i-1})
-r*tau*(1-theta) * V^n_i
*/

coeff3 = r * tau * Space_points[size_Space_points - 1];
DeltaSpaceDown = 1.0 / (Space_points[size_Space_points - 1] - Space_points[siz

diagonal_lhs[size_Space_points - 1] = 1.0 - theta * coeff3 * DeltaSpaceDown +
// Be careful of the index for lower band.
lower_d_lhs[size_Space_points - 2] = theta * coeff3 * DeltaSpaceDown;
diagonal_rhs[size_Space_points - 1] = 1.0 + (1.0 - theta) * coeff3 * DeltaSpace
// Be careful of the index for lower band.
lower_d_rhs[size_Space_points - 2] = -(1.0 - theta) * coeff3 * DeltaSpaceDown;

Mat_lhs = pnl_tridiag_mat_create_from_ptr(size_Space_points, lower_d_lhs, diag
Mat_rhs = pnl_tridiag_mat_create_from_ptr(size_Space_points, lower_d_rhs, diag

Mat_for_non_const_syslin = pnl_tridiag_mat_create(size_Space_points);
pnl_tridiag_mat_clone(Mat_for_non_const_syslin, Mat_lhs);

Solution = pnl_vect_create_from_ptr(size_Space_points, Vector_boundary);
Vector = pnl_vect_create_from_ptr(size_Space_points, Vector_boundary);

for (i = 0; i < NT; i++)
{
    pnl_tridiag_mat_mult_vect_inplace(Vector, Mat_rhs, Solution);
    pnl_tridiag_mat_syslin(Solution, Mat_for_non_const_syslin, Vector);

```

```

// It is necessary to copy Vector in Solution somewhere in the loop.
// So I use mat_syslin instead of mat_syslin_inplace.
pnl_tridiag_mat_clone(Mat_for_non_const_syslin, Mat_lhs);
for (j = 1; j < size_Space_points - 1; j++)
{
    Sortie[j] = pnl_vect_get(Solution, j);
}

// Boundary conditions

Sortie[0] = pnl_vect_get(Solution, 1) - (pnl_vect_get(Solution, 2) - pnl_v
    / (Space_points[2] - Space_points[1]) * (Space_points[1] - Spa
Sortie[size_Space_points - 1] = pnl_vect_get(Solution, size_Space_points
    + (pnl_vect_get(Solution, size_Space_poin
    / (Space_points[size_Space_points - 2] -
    * (Space_points[size_Space_points - 1] -
}
/*Solution in the output pointer*/
for (i = 0; i < size_Space_points; i++)
{
    Sortie[i] = pnl_vect_get(Solution, i);
}

pnl_tridiag_mat_free(&Mat_lhs);
pnl_tridiag_mat_free(&Mat_rhs);
pnl_tridiag_mat_free(&Mat_for_non_const_syslin);
pnl_vect_free(&Solution);
pnl_vect_free(&Vector);
free(upper_d_lhs);
free(diagonal_lhs);
free(lower_d_lhs);
free(upper_d_rhs);
free(diagonal_rhs);
free(lower_d_rhs);
}

void matrix_of_newton_system_new(double *Space_points, int size_Space_points, Pn
    double tau,
    double r, double divid, double sigma_min, doubl
    double theta, double omega, int left_or_right,

```

```

                                PnlTridiagMat *Sortie)
{
    int i;
    double coeff1, coeff2, coeff3, DeltaSpaceUp, DeltaSpaceDown, DeltaSpaceDouble;
    double gamma_local, sigma, alpha, beta;

    for (i = 1; i < size_Space_points - 1; i++)
    {
        DeltaSpaceDown = 1.0 / (Space_points[i] - Space_points[i - 1]);
        DeltaSpaceUp = 1.0 / (Space_points[i + 1] - Space_points[i]);
        DeltaSpaceDouble = 2.0 / (Space_points[i + 1] - Space_points[i - 1]);
        // Computation of the gamma.
        gamma_local = ((pnl_vect_get(Vector, i + 1) - pnl_vect_get(Vector, i)) * D
                        + (pnl_vect_get(Vector, i - 1) - pnl_vect_get(Vector, i)) *
        // Volatility according to gamma.
        if (gamma_local > 0)
        {
            sigma = sigma_min;
        }
        else
        {
            sigma = sigma_max;
        }

        // Central or forward difference scheme.
        if (sigma_min * sigma_min * Space_points[i] * DeltaSpaceDown >= r)
        {
            coeff1 = sigma * sigma * Space_points[i];
            coeff2 = Space_points[i] * DeltaSpaceDouble * tau / 2.0;
            alpha = (coeff1 * DeltaSpaceDown - r) * coeff2;
            beta = (coeff1 * DeltaSpaceUp + r) * coeff2;
        }
        else
        {
            coeff1 = sigma * sigma * Space_points[i];
            alpha = coeff1 * Space_points[i] * tau * DeltaSpaceDown * DeltaSpaceDo
            beta = (coeff1 * DeltaSpaceDouble / 2.0 + r) * Space_points[i] * Delta
        }

        pnl_tridiag_mat_set(Sortie, i, 0, alpha + beta + r * tau);
        pnl_tridiag_mat_set(Sortie, i, -1, -alpha);
    }
}

```



```

    pnl_tridiag_mat_set(Sortie, i, 1, -beta);
}

/* Coefficients should be consistant with boundary conditions. */
/* On boundary S=0. */
pnl_tridiag_mat_set(Sortie, 0, 0, r * tau);
pnl_tridiag_mat_set(Sortie, 0, 1, 0.0);
/* On boundary S=Smax. */
coeff3 = r * tau / (Space_points[size_Space_points - 1] - Space_points[size_Space_points - 2]);
if (left_or_right == -1) // Left member = U^{n+1}
{
    if (theta != 1.0)
    {
// lambda^{n+1}
        pnl_tridiag_mat_set(Sortie, size_Space_points - 1, -1,
                             (-1.0 - coeff3 * Space_points[size_Space_points - 2]));
// mu^{n+1}
        pnl_tridiag_mat_set(Sortie, size_Space_points - 1, 0,
                             (-1.0 + coeff3 * Space_points[size_Space_points - 2]));
    }
    else
    {
// lambda^{n+1}
        pnl_tridiag_mat_set(Sortie, size_Space_points - 1, -1, 0.0);
// mu^{n+1}
        pnl_tridiag_mat_set(Sortie, size_Space_points - 1, 0, 0.0);
    }
}
if (left_or_right == 1) // Right member = U^n
{
    if (theta != 0.0)
    {
// lambda^n
        pnl_tridiag_mat_set(Sortie, size_Space_points - 1, -1,
                             (1.0 - coeff3 * Space_points[size_Space_points - 2]));
// mu^n
        pnl_tridiag_mat_set(Sortie, size_Space_points - 1, 0,
                             (1.0 + coeff3 * Space_points[size_Space_points - 2]));
    }
    else
    {

```

```

// lambda^n
    pnl_tridiag_mat_set(Sortie, size_Space_points - 1, -1, 0.0);
// mu^n
    pnl_tridiag_mat_set(Sortie, size_Space_points - 1, 0, 0.0);
}
}
}
void resolution_pde_sigma_uncertain_implicit(
    double *Space_points, int size_Space_points, double *Vector_boundary,
    double T, int NT,
    double r, double divid, double sigma_min, double sigma_max,
    double theta_global, double omega,
    double max_Iteration_Newton, double tolerance,
    double *Sortie)
{
    int i, j, k;
    double tau, error, difference, theta;
    PnlVect *Solution;
    PnlVect *Old_solution;
    PnlVect *Vector;
    PnlTridiagMat *Working_Matrix;
    PnlTridiagMat *Identity_Matrix;

    /*Initialization*/
    tau = T / (double)NT;

    /*Memory Allocation*/
    Identity_Matrix = pnl_tridiag_mat_create_from_two_double(size_Space_points, 1.
    Working_Matrix = pnl_tridiag_mat_create(size_Space_points);
    Solution = pnl_vect_create_from_ptr(size_Space_points, Vector_boundary);
    Old_solution = pnl_vect_create_from_ptr(size_Space_points, Vector_boundary);
    Vector = pnl_vect_create_from_ptr(size_Space_points, Vector_boundary);
// Rannacher
    theta = theta_global;
    if (NT < 2)
    {
        theta = 1.0; // Explicit.
    }

    for (j = 0; j < NT; j++)
    {

```

```

/* Definition of the M matrix. */
/* With control at S=Smax */
matrix_of_newton_system_new(Space_points, size_Space_points, Solution,
                             tau, r, divid, sigma_min, sigma_max,
                             theta, omega, -1, Working_Matrix);

pnl_tridiag_mat_mult_double(Working_Matrix, -theta);
// Sum with the identity matrix. Result is in the first argument.
pnl_tridiag_mat_plus_tridiag_mat(Working_Matrix, Identity_Matrix);
// Vector <- Working_Matrix * Solution
pnl_tridiag_mat_mult_vect_inplace(Vector, Working_Matrix, Solution);
// Newton iterations.
i = 0;
error = ABS(tolerance) + 1;
pnl_vect_clone(Old_solution, Vector);
while ((i < max_Iteration_Newton) && (error > tolerance))
{
    matrix_of_newton_system_new(Space_points, size_Space_points, Old_solution,
                                tau, r, divid, sigma_min, sigma_max,
                                theta, omega, 1, Working_Matrix);

    /**/
    pnl_tridiag_mat_mult_double(Working_Matrix, (1.0 - theta));
    // Sum with the identity matrix. Result is in the first argument.
    pnl_tridiag_mat_plus_tridiag_mat(Working_Matrix, Identity_Matrix);
    // Solution <- (Working_Matrix)^-1 * Vector
    pnl_tridiag_mat_syslin(Solution, Working_Matrix, Vector);
    // Computation of the error.
    error = 0.0;
    for (k = 0; k < size_Space_points; k++)
    {
        difference = (ABS(pnl_vect_get(Solution, k) - pnl_vect_get(Old_solution, k))
                      / (MAX(1.0, ABS(pnl_vect_get(Solution, k)))));
        if (difference > error)
        {
            error = difference;
        }
    }
    pnl_vect_clone(Old_solution, Solution);
    i++;
}

```

```

    }

    /*Solution in the output pointer*/
    for (i = 0; i < size_Space_points; i++)
    {
        Sortie[i] = pnl_vect_get(Solution, i);
    }

    pnl_tridiag_mat_free(&Identity_Matrix);
    pnl_tridiag_mat_free(&Working_Matrix);
    pnl_vect_free(&Solution);
    pnl_vect_free(&Old_solution);
    pnl_vect_free(&Vector);
}

static int cliquet_scheme(double s, double t,
                          double r, double divid, double sigma_min, double sigma_max,
                          int Kmax, int Jmax, int N, int M,
                          double interpolation, int Number_of_temporal_iteration,
                          double Al, double Ar,
                          double Cl, double Fl, double Cg, double Fg,
                          double theta, double omega,
                          double *ptprice, double *ptdelta)
{
    /*Declarations*/
    int i, j, k, n, m; // Variable index.
    double temp_value; // Temporary variable. Sometimes used to optimize the computation.
    double S_value, P_value; // Temporary variables. Only used to optimize the computation.
    int index_nearest, index_nearest_2, index_nearest_3, index_nearest_4;
    double slow, shigh, plow, phigh;
    double V_slow_plow, V_shigh_plow, V_slow_phigh, V_shigh_phigh;
    double V_temp_Zlow, V_temp_Zhigh;
    double deltaz, R, Rstar, Zplus, h;
    /*Memory Allocation*/
    double *Temporal_points;
    double *Space_points;
    double *New_Space_points;
    double *Previous_Space_points;

```

```

double *Vector_boundary;
double *Z_points;
double **PS_grid;
double ** **Vector_ZPST;
double acc_newton = 0.0000001;
int N_newton = 10;
/*Initial space points*/
double limit_inf = 0.1; // Coefficient between 0 and 1.
// Moreover  $s + \tanh(\text{Space\_points}(N-1)) / \tanh(\text{Space\_points}(0)) * (A1-s)$  near Ar
double limit_sup = 1.0 - tanh((Ar - s) / (A1 - s) * pnl_atanh(-1 + limit_inf))

/*Initialization*/
if ((N % 2) == 0) N++;
if ((Jmax % 2) == 0) Jmax++;

Temporal_points = (double *)malloc((M + 1) * sizeof(double)); // Temporal points

Space_points = (double *)malloc(N * sizeof(double)); // Space points with a ta
Previous_Space_points = (double *)malloc(N * sizeof(double)); // Previous Space points

New_Space_points = (double *)malloc(N * sizeof(double)); // New Space points.

Vector_boundary = (double *)malloc(N * sizeof(double)); // Vector on the boundary

Z_points = (double *)malloc(Kmax * sizeof(double)); // Z points.

PS_grid = (double **)calloc(Jmax, sizeof(double *)); // (P,S) grid.

for (j = 0; j < Jmax; j++)
{
    PS_grid[j] = (double *)calloc(N, sizeof(double));
}

Vector_ZPST = (double ** **)calloc(Kmax, sizeof(double **)); // General vector

for (k = 0; k < Kmax; k++)
{
    Vector_ZPST[k] = (double ** *)calloc(Jmax, sizeof(double **));
}

```

```

    for (j = 0; j < Jmax; j++)
    {
        Vector_ZPST[k][j] = (double **)calloc(N, sizeof(double *));

        for (n = 0; n < N; n++)
        {
            Vector_ZPST[k][j][n] = (double *)calloc((M + 1), sizeof(double));
        }
    }
}

/*Time Step and temporal points*/
for (m = 0; m < M + 1; m++)
{
    Temporal_points[m] = (m * t) / M; // Uniform discretization.
}

// Centered points near s.

for (i = 0; i < N; i++)
{
    Space_points[i] = s + pnl_atanh(-1.0 + limit_inf + ((double)i) / ((double)
        * (2.0 - limit_sup - limit_inf)) / pnl_atanh(limit_inf - 0.5 * (limit_sup + limit_inf)) /
        ;
    Previous_Space_points[i] = Space_points[i] ;
}

/*(P,S) grid construction*/
grid_construction(Space_points, s, PS_grid, Jmax, N);

/* Z grid construction*/
// Discretization of [min(F1,0)*M,C1*M]....
deltaz = (C1 * M - MIN(F1, 0) * M) / (Kmax - 1.0);
Z_points[0] = MIN(F1, 0);
for (k = 1; k < Kmax; k++)
{

```

```

    Z_points[k] = Z_points[k - 1] + deltaz;
}

/*Definition of the terminal value.
It depends only on Z variable which contains all the information.*/
for (k = 0; k < Kmax; k++) // Loop on Z variable.
{
    temp_value = MAX(Fg, MIN(Cg, Z_points[k]));
    for (j = 0; j < Jmax; j++) // Loop on P variable.
    {
        for (n = 0; n < N; n++) // Loop on space variable S.
        {
            Vector_ZPST[k][j][n][M] = temp_value; // Terminal value.
        }
    }
}

/*General temporal loop*/
for (m = M - 1; m > -1; m--)
{
    /*Loop on the Z variable*/
    for (k = 0; k < Kmax; k++)
    {
        /*Loop on the P variable*/
        for (j = 0; j < Jmax; j++)
        {
            /*Loop on the space variable S*/
            for (n = 0; n < N; n++)
            {
                S_value = PS_grid[j][n];
                P_value = Previous_Space_points[j];
                // We want to compute the value of the option with the jump co
                // Given a value of %Z of Z variable, we compute the value of
                //  $V(\%S, t^-, \%P, \%Z) = V(\%S, t^+, \%S, \%Z + \max(Fl, \min$ 
                // but since the value of the option V is only defined on a fi
                // we need some interpolations. Here %S = PS_grid[j][n] and %P
                // First, we should compute the concerned index in the Z-grid
                R = (S_value / P_value) - 1.0;
                Rstar = MAX(Fl, MIN(Cl, R));
                Zplus = Z_points[k] + Rstar;
                index_nearest = nearest(Z_points, Kmax, Zplus, 0);
            }
        }
    }
}

```

```

// Rstar is positive, so the index is actually greater than k
// But if Fl is negative, this is not the case.
// If the Z-grid is uniform then the index can be computed exp
// index_nearest = (int floor) (k + Rstar/SIZE_STEP_IN_Z_GRID)
if (index_nearest == Kmax - 1)
{
    // if index_nearest = Kmax-1, we are on the boundary.
    // we will do a linear extrapolation with the two last Z v
    // This extrapolation will actually be an interpolation wi
    // but this is not a problem and it is sufficient to do :
    index_nearest--;
}
// The actualization P+ = S needs an interpolation.
// We need the value V ( %S, t^+, %S , Zplus), so we study the
if (interpolation == 0) // xy-interpolation
{
    // First, we look for the Plow and Phigh values of the (%S
    index_nearest_2 = nearest_lower(Previous_Space_points, Jma
    // Take care of the condition on the boundary
    // if S_value < Previous_Space_points_min i.e. index_neare
    // or if S_value > Previous_Space_points_max i.e. index_ne
    // In these cases, the new values are not given by an inte
    // but just by the value in the extremal points
    // "according to a similarity condition".
    if (index_nearest_2 == -1)
    {
        // Looking for the index of the nearest S value of Pre
        // This should be (int)((N-1)/2) for the scaled grid.
        index_nearest_3 = nearest(PS_grid[0], N, Previous_Spac
        V_temp_Zlow = interpolation_linear(
            PS_grid[0][index_nearest_3], PS_grid[0]
            Vector_ZPST[index_nearest][0][index_ne
            Vector_ZPST[index_nearest][0][index_ne
            Previous_Space_points[0]);
        V_temp_Zhigh = interpolation_linear(
            PS_grid[0][index_nearest_3], PS_grid[0]
            Vector_ZPST[index_nearest + 1][0][ind
            Vector_ZPST[index_nearest + 1][0][ind
            Previous_Space_points[0]);
    }
}
else

```



```

{
    if (index_nearest_2 == N - 1)
    {
        // Looking for the index of the nearest S value of
        // This should be (int)((N-1)/2) for the scaled gr
        index_nearest_3 = nearest(PS_grid[Jmax - 1], N, Pr
        V_temp_Zlow = interpolation_linear(
            PS_grid[Jmax - 1][index_nearest_3]
            Vector_ZPST[index_nearest][Jmax -
            Vector_ZPST[index_nearest][Jmax -
            Previous_Space_points[Jmax - 1]);
        V_temp_Zhigh = interpolation_linear(
            PS_grid[0][index_nearest_3], PS_g
            Vector_ZPST[index_nearest + 1][0]
            Vector_ZPST[index_nearest + 1][0]
            Previous_Space_points[0]);
    }
    else
    {
        // The Plow and Phigh values of the (%S,%S) point
        // Now, we look for the Slow and Shigh values of t
        index_nearest_3 = nearest(PS_grid[index_nearest_2]
        slow = Space_points[index_nearest_2];
        plow = slow; // Xy-interpolation
        shigh = Space_points[index_nearest_2 + 1];
        phigh = shigh; // Interpolation on the diagonal.

        // The index on the PS-grid in order to do the int
        // corresponds to the index j=index_nearest_2 and
        // n = index such that Space_points[n] = sstar val
        // of the PS_grid.
        // Indeed, we need a point (X,X) = (PS_grid[j][n],
        // := (S_j * S_n / Sstar, Space_points[index_neares
        // with S_j = Space_points[index_nearest_2] so S_
        // Actually, this index is (N-1)/2 !
        V_slow_plow = Vector_ZPST[index_nearest][index_nea
            [(int)((N - 1) / 2)][m + 1];
        // V_shigh_plow is useless in the diagonal interpo
        V_shigh_plow = Vector_ZPST[index_nearest][index_ne
            [(int)((N - 1) / 2)][m + 1];
        // V_slow_phigh is useless in the diagonal interpo

```

```

V_slow_phigh = Vector_ZPST[index_nearest][index_nearest + 1][m + 1];
V_shigh_phigh = Vector_ZPST[index_nearest][index_nearest + 1][m + 1];

V_temp_Zlow = interpolation_diagonal(slow, shigh,
                                     V_slow_plow,
                                     PS_grid[j][n]);

V_slow_plow = Vector_ZPST [index_nearest + 1][index_nearest + 1][m + 1];
// V_shigh_plow is useless in the diagonal interpolation
V_shigh_plow = Vector_ZPST [index_nearest + 1][index_nearest + 1][m + 1];
// V_slow_phigh is useless in the diagonal interpolation
V_slow_phigh = Vector_ZPST [index_nearest + 1][index_nearest + 1][m + 1];
V_shigh_phigh = Vector_ZPST [index_nearest + 1][index_nearest + 1][m + 1];

V_temp_Zhigh = interpolation_diagonal(slow, shigh,
                                     V_slow_plow,
                                     PS_grid[j][n]);

}

Vector_ZPST[k][j][n][m] = interpolation_linear(
    Z_points[index_nearest], Z_points[index_nearest + 1],
    V_temp_Zlow, V_temp_Zhigh, Zplow, Zphigh);
}
else // Diagonal interpolation
{
    // First, we look for the Plow and Phigh values of the (%S)
    index_nearest_2 = nearest_lower(Previous_Space_points, Jmax);
    // Take care of the condition on the boundary
    // if S_value < Previous_Space_points_min i.e. index_nearest_2 == -1
    // or if S_value > Previous_Space_points_max i.e. index_nearest_2 == N
    // In these cases, the new values are not given by an interpolation
    // but just by the value in the extremal points
    // "according to a similarity condition".
    if (index_nearest_2 == -1)

```

```

{
    // Looking for the index of the nearest S value of Pre
    // This should be (int)((N-1)/2) for the scaled grid.
    index_nearest_3 = nearest(PS_grid[0], N, Previous_Space_points[0]);
    V_temp_Zlow = interpolation_linear(
        PS_grid[0][index_nearest_3], PS_grid[0][index_nearest_3 + 1],
        Vector_ZPST[index_nearest][0][index_nearest_3],
        Vector_ZPST[index_nearest][0][index_nearest_3 + 1],
        Previous_Space_points[0]);
    V_temp_Zhigh = interpolation_linear(
        PS_grid[0][index_nearest_3], PS_grid[0][index_nearest_3 + 1],
        Vector_ZPST[index_nearest + 1][0][index_nearest_3],
        Vector_ZPST[index_nearest + 1][0][index_nearest_3 + 1],
        Previous_Space_points[0]);
}
else
{
    if (index_nearest_2 == N - 1)
    {
        // Looking for the index of the nearest S value of Pre
        // This should be (int)((N-1)/2) for the scaled grid.
        index_nearest_3 = nearest(PS_grid[Jmax - 1], N, Previous_Space_points[Jmax - 1]);
        V_temp_Zlow = interpolation_linear(
            PS_grid[Jmax - 1][index_nearest_3], PS_grid[Jmax - 1][index_nearest_3 + 1],
            Vector_ZPST[index_nearest][Jmax - 1][index_nearest_3],
            Vector_ZPST[index_nearest][Jmax - 1][index_nearest_3 + 1],
            Previous_Space_points[Jmax - 1]);
        V_temp_Zhigh = interpolation_linear(
            PS_grid[0][index_nearest_3], PS_grid[0][index_nearest_3 + 1],
            Vector_ZPST[index_nearest + 1][0][index_nearest_3],
            Vector_ZPST[index_nearest + 1][0][index_nearest_3 + 1],
            Previous_Space_points[0]);
    }
    else
    {
        // Looking for the index of a diagonal point in the grid
        index_nearest_3 = nearest(PS_grid[index_nearest_2], N, Previous_Space_points[index_nearest_2]);
        if (index_nearest_3 == -1)
        {
            index_nearest_3 = 0;
        }
    }
}

```

```

    }
    if (index_nearest_3 == N - 1)
    {
        index_nearest_3 = N - 2;
    }
    // Looking for the index of the nearest higher S v
    index_nearest_4 = nearest(PS_grid[index_nearest_2
        Previous_Space_points[in

// We have find  $\text{plow} < \%S < \text{phigh}$  and  $\text{slow} < \%S < \text{shigh}$ 
slow = PS_grid[index_nearest_2][index_nearest_3];
plow = Previous_Space_points[index_nearest_2];
shigh = PS_grid[index_nearest_2 + 1][index_nearest_3];
phigh = Previous_Space_points[index_nearest_2 + 1][index_nearest_3];

//////////

// In the case of the scaled grid,
// the index on the PS-grid in order to do the interpolation
// corresponds to the index  $j = \text{index\_nearest\_2}$  and
//  $n = \text{index}$  such that  $\text{Space\_points}[n] = \text{sstar\_value}$ 
// of the PS_grid.
// Indeed, we need a point  $(X,X) = (\text{PS\_grid}[j][n], \text{Space\_points}[n])$ 
//  $:= (S_j * S_n / S_{\text{star}}, \text{Space\_points}[n])$ 
// with  $S_j = \text{Space\_points}[\text{index\_nearest\_2}]$  so  $S_n = \text{sstar\_value}$ 
// Actually, this index is  $(N-1)/2$  !

//////////

V_slow_plow = Vector_ZPST [index_nearest][index_nearest_3][m + 1];
// V_shigh_plow is useless in the diagonal interpolation
V_shigh_plow = 0;
// V_slow_phigh is useless in the diagonal interpolation
V_slow_phigh = 0;
V_shigh_phigh = Vector_ZPST [index_nearest][index_nearest_4][m + 1];

V_temp_Zlow = interpolation_diagonal(slow, shigh,
    V_slow_plow,
    PS_grid[j][n]

V_slow_plow = Vector_ZPST [index_nearest + 1][index_nearest_3][m + 1];

```

```

// V_shigh_plow is useless in the diagonal interpolation
V_shigh_plow = 0;
// V_slow_phigh is useless in the diagonal interpolation
V_slow_phigh = 0;
V_shigh_phigh = Vector_ZPST [index_nearest + 1] [index_nearest_4] [m + 1];

V_temp_Zhigh = interpolation_diagonal(slow, shigh,
                                     V_slow_plow,
                                     PS_grid[j] [n])
    }
}
Vector_ZPST[k] [j] [n] [m] = interpolation_linear(
    Z_points[index_nearest], Z_points[index_nearest_4],
    V_temp_Zlow, V_temp_Zhigh, Zpl)
}
} /*Loop on the S variable*/
} /*Loop on the P variable*/
} /*Loop on the Z variable*/

/*Loop on the Z variable*/
for (k = 0; k < Kmax; k++)
{
    /*Loop on the P variable*/
    for (j = 0; j < Jmax; j++)
    {
        /*Redefinition of the space points for the FD scheme*/
        for (n = 0; n < N; n++)
        {
            New_Space_points[n] = PS_grid[j] [n];
        }
        /*Definition of the value on the boundary.
        It corresponds to the value in t_m^- i.e. the values for the temperature
        after the jump conditions.*/
        for (n = 0; n < N; n++)
        {
            Vector_boundary[n] = Vector_ZPST[k] [j] [n] [m];
        }

        /*Call of the procedure for the FD scheme.*/
        if (sigma_min == sigma_max)

```

```

        {
            // sigma <- sigma_min
            resolution_pde_sigma_constant(New_Space_points, N, Vector_boun
                Temporal_points[m + 1] - Tempora
                r, divid, sigma_min, theta, omeg
                Vector_boundary);
        }
    else
    {
        resolution_pde_sigma_uncertain_implicit(New_Space_points, N, V
            Temporal_points[m + 1]
            r, divid, sigma_min, s
            N_newton, acc_newton,
            Vector_boundary);
    }

    /*Record of the solution in the big array Vector_ZPST.*/
    for (n = 0; n < N; n++)
    {
        Vector_ZPST[k][j][n][m] = Vector_boundary[n];
    }
    } /*Loop on the P variable*/
} /*Loop on the Z variable*/
} /*General temporal loop*/

/*Find the good multiple index (k,j,n,m) for the price and the delta*/
// Temporary minimal value and its index in order to find the index near the 0
index_nearest = nearest(Z_points, Kmax, 0, 0);

if (Z_points[index_nearest] > 0)
{
    index_nearest = index_nearest - 1; // Point under 0 value.
}
// Index of the Z value just before 0.
if (index_nearest < 0)
{
    index_nearest = 0; //
}

index_nearest_2 = nearest(Previous_Space_points, Jmax, s, 0);

```

```

index_nearest_3 = nearest(PS_grid[index_nearest_2], N,
                          Previous_Space_points[index_nearest_2], 0);
index_nearest_4 = nearest(PS_grid[index_nearest_2 + 1], N,
                          Previous_Space_points[index_nearest_2 + 1], 0);
// We have find plow < s < phigh and slow < s < shigh.
slow = PS_grid[index_nearest_2][index_nearest_3];
plow = Previous_Space_points[index_nearest_2];
shigh = PS_grid[index_nearest_2 + 1][index_nearest_4];
phigh = Previous_Space_points[index_nearest_2 + 1];

V_slow_plow = Vector_ZPST [index_nearest][index_nearest_2]
               [index_nearest_3][m + 1];
V_shigh_phigh = Vector_ZPST [index_nearest][index_nearest_2 + 1]
               [index_nearest_4][m + 1];
m = 0;
/*Price*/
// We do an interpolation if the Z value is not equal to 0.
if (Z_points[index_nearest] == 0.)
{
    *ptprice = Vector_ZPST[index_nearest][index_nearest_2][index_nearest_3][m]
}
else
{
    *ptprice = interpolation_linear(Z_points[index_nearest], Z_points[index_n
                                   Vector_ZPST[index_nearest][index_nearest_
                                   Vector_ZPST[index_nearest + 1][index_near

}
/*Delta*/
// We do an interpolation if the Z value is not equal to 0.
h = (PS_grid[index_nearest_2][index_nearest_3 + 1] - PS_grid[index_nearest_2][
if (Z_points[index_nearest] == 0.)
{
    *ptdelta = (Vector_ZPST[index_nearest][index_nearest_2][index_nearest_3 +
               - Vector_ZPST[index_nearest][index_nearest_2][index_nearest_3

}
else
{
    *ptdelta = (interpolation_linear(Z_points[index_nearest], Z_points[index_n
                                   Vector_ZPST[index_nearest][index_nearest_
                                   Vector_ZPST[index_nearest + 1][index_near

```

```

        interpolation_linear(Z_points[index_nearest], Z_points[index_n
                                Vector_ZPST[index_nearest][index_nearest_
                                Vector_ZPST[index_nearest + 1][index_near

    ) / h;
}

/*Memory Desallocation*/
free(Temporal_points);

free(Space_points);

free(Previous_Space_points);

free(New_Space_points);

free(Vector_boundary);

free(Z_points);
Z_points = NULL;

for (j = Jmax - 1; j > -1; j--)
{
    free(PS_grid[j]);
}
free(PS_grid);

for (k = Kmax - 1; k > -1; k--)
{
    for (j = Jmax - 1; j > -1; j--)
    {
        for (n = N - 1; n > -1; n--)
        {
            free(Vector_ZPST[k][j][n]);
        }
        free(Vector_ZPST[k][j]);
    }
    free(Vector_ZPST[k]);
}
free(Vector_ZPST);

```



```

    return 0;
}

static int UVM(int t, double Fg, double Cg, double Fl, double Cl, double s, double r, double divid,
               double sigma_min, double sigma_max, double Kmax, int Jmax, int N, int M, int interpolation,
               int Number_of_temporal_iterations, double Al, double Ar, double Cl, double Fl, double Cg, double Fg,
               double theta, double omega, double ptprice, double ptdelta)
{
    int Jmax;
    int M = pnl_iround(t); // M = Number of temporal iterations.
    double interpolation = 1; // 0 for xy-interpolation, else diagonal interpolation.

    double Al = s / 100.0; // Al = Minimal value of the space domain.
    double Ar = s * 9.0; // Ar = Maximal value of the space domain.

    // Parameters which should NOT be modified by the user.
    double theta = 0.5; // Theta of the theta scheme.
    double omega = 1.0; // Between 0 and 2.

    Jmax = N;
    cliquet_scheme(s, t, r, divid, sigma_min, sigma_max,
                  Kmax, Jmax, N, M, interpolation, Number_of_temporal_iterations,
                  Al, Ar, Cl, Fl, Cg, Fg,
                  theta, omega, ptprice, ptdelta);

    return OK;
}

int CALC(FD_UVM)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r, divid;
    double Fg, Cg, Fl, Cl;

    Fg = (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE;
    Cg = (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[1].Val.V_PDOUBLE;
    Fl = (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[2].Val.V_PDOUBLE;
    Cl = (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[3].Val.V_PDOUBLE;

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

```

```

        return UVM(ptOpt->Maturity.Val.V_PINT, Fg, Cg, Fl, Cl, ptMod->S0.Val.V_PDOUBLE,
                    r, divid, ptMod->sigmamin.Val.V_PDOUBLE, ptMod->sigmamax.Val.V_PDOU
                    Met->Par[0].Val.V_INT, Met->Par[1].Val.V_INT, Met->Par[2].Val.V_INT
                    &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE));
    }
    //ptOpt->Fg.Val.V_PDOUBLE,ptOpt->Cg.Val.V_PDOUBLE,ptOpt->Fl.Val.V_PDOUBLE,ptOpt-
    static int CHK_OPT(FD_UVM)(void *Opt, void *Mod)
    {
        if ((strcmp(((Option *)Opt)->Name, "Cliquet") == 0))
            return OK;
        return WRONG;
    }

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;

        Met->Par[0].Val.V_INT2 = 100;
        Met->Par[1].Val.V_INT2 = 64;
        Met->Par[2].Val.V_INT2 = 20;

    }

    return OK;
}

PricingMethod MET(FD_UVM) =
{
    "FD_UVM",
    {"TimeStepforYear", INT2, {100}, ALLOW} , {"SpaceStep S", INT2, {100}, ALLOW},
    CALC(FD_UVM),
    { {"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(FD_UVM),
    CHK_ok,
    MET(Init)
}

```

};