

## [Help](#)

```
/**
 * Stage Cermics 2015
 *
 * @file pricer.c
 * @brief The Stochastic Grid Bundling Method.
 *
 * Implementation of the method described in "The Stochastic Grid Bundling Method:
 * Efficient Pricing of Bermudan Options and their Greeks" by Shashi Jain and
 * Cornelis W. Oosterlee.
 *
 * @date 17.08.15
 */
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_basis.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_matrix.h"
#include "cmath"

using namespace std;

extern "C" {
#include "
href../../common/math/clustering_h_src.pdfmath/clustering.h"
#include "
href../../mod/bsnd/bsnd_stdnd/bsnd_stdnd_h_src.pdfbsnd_stdnd.h"
}

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2016+2) //The "#els
extern "C" {
static int CHK_OPT(MC_JainOosterlee1ND)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_JainOosterlee1ND)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
```

```

}
#else
/*! \ mainpage The Stochastic Grid Bundling Method
*
* \ section intro_sec Main Page of Documentation
* Description of the method and instructions for the use of the pricer.
*
* \ section desc_sec Description of the method SGBM
* \ subsection step1 1) Generating grid points
* \ subsection step2 2) Option value at terminal time
* \ subsection step3 3) Bundling
* \ subsection step4 4) Mapping high-dimensional state space to a low-dimension
* \ subsection step5 5) Computing the continuation and option values at  $t_{m-1}$ 
*
* \ section exec_sec Use of the pricer
* PNL must be installed<br>
* To compile, execute:
* @code make @endcode
* To execute the pricer:
* @code ./pipo @endcode
*/

/**
* @brief Clustering methods.
*/
typedef enum {
    KMEANS = 1,
    RB      = 2,
    RBRSS   = 3,
} ClusteringMethods;

/**
* @brief Possible basis functions to build regression matrix.
*/
typedef enum {
    GEOMETRIC = 1,
    ARITHMETIC = 2,
    MONOMIAL = 3,
} RegressionMatrixTypes;

/**

```

```

    * @brief Option types.
    */
typedef enum {
    CALL = 1,
    PUT  = 2,
} OptionTypes;

/**
 * @brief Option styles.
 */
typedef enum {
    EUROPEAN = 1,
    AMERICAN = 2,
} ExerciseTypes;

/**
 * @brief Instrument types.
 */
typedef enum {
    SINGLE = 1,
    BASKET = 2,
} InstrumentTypes;

// generic instrument
static InstrumentTypes InstrumentType = SINGLE;
static ExerciseTypes   ExerciseType = EUROPEAN;
static int             dim = 1;
static double          T = 1.;
static int NumExerciseDates = 0;
static double* ExerciseDates = NULL;
static double* CDividendR = NULL;

// vanilla
static RegressionMatrixTypes RegressMatType = GEOMETRIC;
static OptionTypes OptionType = PUT;
static double Strike = 40.;
static double* S0 = NULL;
static double* sigma = NULL;
static double r = 0.06;
static double discount = 0.;

```

```

// basket
static double rho = 0.;
static PnlMat* rhoMat = NULL;
static PnlMat* SigmaMat = NULL;

// model
static int NRepl = 5;
static int Nsteps = 4;
static double* DiscretDates = NULL;
static int numDates = 0;
static double* Dates = NULL;

// pricer
static int ClustMethod = KMEANS;
static int K = 2;
static int H = 3;

/**
 * @brief Set exercise dates.
 *
 * @param [in] numED Indicate time step between exercise dates : T/numED.
 */
static void setExerciseDates(int numED)
{
    ExerciseDates = (double*) malloc((numED+1)*sizeof(double));
    if (ExerciseDates == NULL)
        abort();
    int i;
    for (i = 0; i <= numED; i++)
        ExerciseDates[i] = i*(T/numED);
}

/**
 * @brief Create the correlation matrix.
 */
static void setRhoMat()
{
    rhoMat = pnl_mat_create_from_scalar(dim, dim, rho);
}

```

```

    pnl_mat_set_diag(rhoMat, 1., 0);
}

/**
 * @brief Create Sigma matrix by doing Cholesky decomposition.
 */
static void setSigmaMat()
{
    int i, k;
    SigmaMat = pnl_mat_create(dim, dim);

    for (i = 0; i < dim; i++)
        for (k = 0; k < dim; k++)
            MLET(SigmaMat, i, k) = sigma[i]*sigma[k]*MGET(rhoMat, i, k);

    if (pnl_mat_chol(SigmaMat) == FAIL)
        abort();
}

/**
 * @brief Free the correlation and Sigma matrices.
 */
static void freeRhoSigmaMats()
{
    if (rhoMat != NULL)
        pnl_mat_free(&rhoMat);
    if (SigmaMat != NULL)
        pnl_mat_free(&SigmaMat);
}

/**
 * @brief Compute the exercise value.
 *
 * @param [in] SPaths Simulated stochastic paths.
 * @param [in] step Time step in which it is computed the exercise value.
 * @param [in, out] ExerciseValue Computed exercise value.
 *
 * Memory must be previously reserved.
 */

```

```

static void getExerciseValue(PnlMat** SPaths, int step, PnlVect* ExerciseValue)
{
    assert(SPaths != NULL);
    assert(ExerciseValue != NULL);

    // Simulated stochastic paths at time step 'step'
    PnlMat* SPathsAux = pnl_mat_copy(SPaths[step]);

    if (RegressMatType == GEOMETRIC)
        pnl_mat_cumprod(SPathsAux, 'c');
    else if (RegressMatType == ARITHMETIC)
        pnl_mat_cumsum(SPathsAux, 'c');
    pnl_mat_get_col(ExerciseValue, SPathsAux, (SPathsAux->n)-1);

    int i;
    double aux;
    if (OptionType == CALL) // Call payoff
    {
        for (i = 0; i < ExerciseValue->size; i++)
        {
            if (RegressMatType == GEOMETRIC)
                aux = pow(GET(ExerciseValue, i), 1. / ((double) SPathsAux->n))
                    - Strike;
            else
                aux = GET(ExerciseValue, i)/((double) SPathsAux->n) - Strike;
            if (aux < 0)
                LET(ExerciseValue, i) = 0.;
            else
                LET(ExerciseValue, i) = aux;
        }
    }
    else if (OptionType == PUT) // Put payoff
    {
        for (i = 0; i < ExerciseValue->size; i++)
        {
            if (RegressMatType == GEOMETRIC)
                aux = Strike -
                    pow(GET(ExerciseValue, i), 1. / ((double) SPathsAux->n));
            else
                aux = Strike - GET(ExerciseValue, i)/((double) SPathsAux->n);
        }
    }
}

```

```

    if (aux < 0)
    LET(ExerciseValue, i) = 0.;
    else
    LET(ExerciseValue, i) = aux;
}
}

```

```

    pnl_mat_free(&SPathsAux);

```

```

    return;
}

```

```

/**
 * @brief Compute the continuation value.
 *
 * @param [in] SPaths Simulated stochastic paths.
 * @param [in] i Time step to which we actualise.
 * @param [in, out] CashFlows At entry, value at time step j.
 *                                     At exit, continuation value.
 * @param [in] j Time step from which we actualise.
 * @param [in] dat Dates.
 */
static void getContinuationValue(PnlMat** SPaths, int i, PnlVect* CashFlows,
                                int j, const double* dat)
{
    assert(SPaths != NULL);
    assert(CashFlows != NULL);

    discount = exp(-r*(dat[j] - dat[i]));
    pnl_vect_mult_scalar(CashFlows, discount);

    return;
}

```

```

/**
 * @brief Set discretization dates.
 *

```

```

    * @param [in] numDD Indicate time step between discretization dates :  $T/\text{numDD}$ .
    */
static void setDiscretDates(int numDD)
{
    DiscretDates = (double*) malloc((numDD+1)*sizeof(double));
    int i;
    for (i = 0; i <= numDD; i++)
        DiscretDates[i] = i*(T/numDD);
}

/**
 * @brief Compare two objects.
 *
 * @param [in] arg1 Object 1.
 * @param [in] arg2 Object 2.
 *
 * @retval -1 Second object bigger.
 * @retval 1 First object bigger.
 * @retval 0 Objects are equals.
 */
static int comparer(const void *arg1, const void *arg2)
{
    if (*(double*) arg1 < *(double*) arg2)
        return -1;
    else if (*(double*) arg1 > *(double*) arg2)
        return 1;
    else
        return 0;
}

/**
 * @brief Compose the ensemble of exercise, discretization and coupon dates
 *        without repetitions.
 */
static void setDates()
{
    setDiscretDates(Nsteps);
    if (NumExerciseDates != 0)
    {

```



```

        setExerciseDates(NumExerciseDates);
        numDates = Nsteps + NumExerciseDates;
    }
    else
        numDates = Nsteps + 1;

    Dates = (double*) malloc(numDates*sizeof(double));
    if (Dates == NULL)
        abort();
    memcpy(Dates, DiscretDates, (Nsteps+1)*sizeof(double));
    if (NumExerciseDates != 0)
    {
        memcpy(&Dates[Nsteps+1], &ExerciseDates[1],
            (NumExerciseDates-1)*sizeof(double));

        // Quick-Sort
        qsort(Dates, numDates, sizeof(double), comparer);
        // remove repetitions
        int i;
        int count = 1;
        for (i = 1; i < numDates; i++)
        {
            if (Dates[i] != Dates[i-1])
            {
                Dates[count] = Dates[i];
                count++;
            }
        }
        numDates = count;
    }

    return;
}

/**
 * @brief Generate the stochastic paths.
 *
 * @param [out] SPaths Simulated stochastic paths.
 * @param [in] generator Random generator.
 */

```

```

static void getTrajectories(PnlMat** SPaths, PnlRng* generator)
{
    assert(SPaths != NULL);
    assert(generator != NULL);

    /// STEP I : Generating grid points (sample paths)
    int i, j;
    double dt;

    if (InstrumentType == SINGLE)
    {
        double auxValue;
        PnlMat* SPathsAux = pnl_mat_new();
        PnlVect* initialV = pnl_vect_create_from_scalar(NRepl, log(S0[0]));
        pnl_mat_rng_normal(SPathsAux, NRepl, numDates, generator);
        pnl_mat_set_col(SPathsAux, initialV, 0);

        for (j = 1; j < numDates; j++)
        {
            dt = Dates[j] - Dates[j-1];
            for (i = 0; i < NRepl; i++)
            {
                auxValue = (r - CDividendR[dim-1] - 0.5*SQR(sigma[0]))*dt +
                    sigma[0]*sqrt(dt)*MGET(SPathsAux, i, j);
                MLET(SPathsAux, i, j) = auxValue;
            }
        }
        pnl_mat_cumsum(SPathsAux, 'c');

        for (j = 0; j < numDates; j++)
        {
            pnl_mat_resize(SPaths[j], NRepl, dim);
            for (i = 0; i < NRepl; i++)
            {
                MLET(SPaths[j], i, dim-1) = exp(MGET(SPathsAux, i, j));
            }
        }

        pnl_vect_free(&initialV);
        pnl_mat_free(&SPathsAux);
    }
}

```

```

else
{
    int k, l;
    double depBrow;
    double SigmaMatValue = 0.;
    PnlMat* SPathsAux = pnl_mat_create(dim, numDates);
    PnlMat* NormalMat = NULL;

    for (j = 0; j < NRepl; j++)
    {
        for (l = 0; l < dim; l++)
        {
            MLET(SPathsAux, l, 0) = log(S0[l]);
        }
        NormalMat = pnl_mat_new();
        pnl_mat_rng_normal(NormalMat, dim, numDates, generator);
        for (i = 1; i < numDates; i++)
        {
            dt = Dates[i] - Dates[i-1];
            for (k = 0; k < dim; k++)
            {
                depBrow = 0.0;
                for (l = 0; l < dim; l++)
                {
                    SigmaMatValue = MGET(SigmaMat, k, l);
                    depBrow += SigmaMatValue * MGET(NormalMat, l, i);
                }
                depBrow = depBrow / sigma[k];
                MLET(SPathsAux, k, i) = (r - CDividendR[k] -
                                         0.5*SQR(sigma[k]))*dt +
                                         sigma[k]*sqrt(dt)*depBrow;
            }
        }
        pnl_mat_cumsum(SPathsAux, 'c');

        for (i = 0; i < numDates; i++)
        {
            if (j == 0)
                pnl_mat_resize(SPaths[i], NRepl, dim);
            for (k = 0; k < dim; k++)
                MLET(SPaths[i], j, k) = exp(MGET(SPathsAux, k, i));
        }
    }
}

```

```

        }

        pnl_mat_free(&NormalMat);
    }

    pnl_mat_free(&SPathsAux);
}

return;
}

/**
 * @brief Apply geometric mean in each row.
 *
 * @param [in] m Matrix of real numbers.
 * @param [out] h Vector with geometric mean of each row.
 *
 * @see recursiveBifurcationRSS
 */
static void geometricMean(const PnlMat* m, PnlVect* h)
{
    assert(m != NULL);
    assert(h != NULL);

    int i;
    pnl_mat_prod_vect(h, m, 'c');
    for (i = 0; i < h->size; i++)
        LET(h, i) = pow(GET(h, i), 1./((double) m->n));

    return;
}

/**
 * @brief Apply arithmetic mean in each row.
 *
 * @param [in] m Matrix of real numbers.
 * @param [out] h Vector with arithmetic mean of each row.
 *
 * @see recursiveBifurcationRSS

```

```

    */
static void arithmeticMean(const PnlMat* m, PnlVect* h)
{
    assert(m != NULL);
    assert(h != NULL);

    pnl_mat_sum_vect(h, m, 'c');
    pnl_vect_div_scalar(h, (double) m->n);

    return;
}

/* R^(dim) --> R function to apply to paths to map the high-dimensional state
   space onto a single-dimensional space. */
static void (*mappingF_ptr) (const PnlMat*, PnlVect*) = arithmeticMean;

/**
 * @brief Compute a clustering with a chosen method.
 *
 * @param [in] SPaths Matrix to cluster.
 * @param [out] SPaths_Clusterized Vector indicating which cluster correspond to
 *                                     each row of SPaths.
 * @param [in, out] KMctrs Centres for k-means clustering algorithm.
 *                                     Optional, can be NULL.
 * @param [in] rng Random generator.
 *
 * @retval OK Clustering done correctly.
 * @retval ERR An error occurred.
 */
static int clustering(const PnlMat* SPaths, int* SPaths_Clusterized,
                    PnlMat** KMctrs, PnlRng* rng)
{
    assert(SPaths != NULL);
    assert(SPaths_Clusterized != NULL);
    assert(rng != NULL);

    if (ClustMethod == KMEANS)
    {
        Model_kmeans mod;

```

```

    if (*KMctrs == NULL)
    {
        if (kmeans(SPaths->array, SPaths->m, SPaths->n, K, SPaths_Clusterize
                    &mod, NULL, MAX_ITERATIONS, rng) == ERR)
        {
            freeModel_kmeans(&mod);
            return ERR;
        }
        *KMctrs = pnl_mat_create_from_ptr(K, SPaths->n, mod.centroids);
    }
    else
    {
        if (kmeans(SPaths->array, SPaths->m, SPaths->n, K, SPaths_Clusterize
                    &mod, (*KMctrs)->array, 1, rng) == ERR)
        {
            freeModel_kmeans(&mod);
            return ERR;
        }
    }
    freeModel_kmeans(&mod);
}
else if (ClustMethod == RB)
{
    // 2^(Ncols*p) number of clusters for recursive bifurcation
    int p1 = (int) (log(K)/(log(2.)*(SPaths->n)));
    if (recursiveBifurcation(SPaths, p1, SPaths_Clusterized) == ERR)
        return ERR;
}
else if (ClustMethod == RBRSS)
{
    // 2^p number of clusters for recursive bifurcation RSS
    int p2 = (int) (log(K)/log(2.));
    if (recursiveBifurcationRSS(SPaths, p2, SPaths_Clusterized, mappingF_ptr
        == ERR)
        return ERR;
}
else // It is not a valid clustering method
    return ERR;

return OK;
}

```

```

/**
 * @brief Search the index of an integer vector which values are equal to value.
 *
 * @param [out] index Pointer to the array with index found.
 * @param [in] value Integer value to compare with.
 * @param [in] vect Integer vector.
 * @param [in] len Vector length.
 *
 * @return Number of index found.
 */
static int findIndex_EqualValueInt(int** index, int value, const int* vect, int
{
    assert(vect != NULL);

    int i;
    int count = 0;
    int allocatedTam = 0;
    int futurTam = 400;
    int* indexAux;

    for (i = 0; i < len; i++)
    {
        if (vect[i] == value)
        {
            if (count >= allocatedTam)
            {
                allocatedTam = futurTam;
                indexAux = (int*) realloc(*index, allocatedTam*sizeof(int));
                if (indexAux == NULL)
                    abort();
                *index = indexAux;
                futurTam = futurTam*2;
            }
            (*index)[count] = i;
            count++;
        }
    }

    return count;
}

```

```
}
```

```
/**
 * @brief Search the index corresponding to vector1 values greater than vector2
 *
 * @param [out] index Pointer to the array with index found.
 * @param [in] vect1 Vector 1.
 * @param [in] vect2 Vector 2.
 *
 * @return Number of index found.
 */
static int findIndex_GreaterValue(int** index, const PnlVect* vect1,
                                   const PnlVect* vect2)
{
    assert(vect1 != NULL);
    assert(vect2 != NULL);

    int i;
    int count = 0;
    int allocatedTam = 0;
    int futurTam = 100;
    int* indexAux;

    for (i = 0; i < vect1->size; i++)
    {
        if (GET(vect1, i) > GET(vect2, i))
        {
            if (count >= allocatedTam)
            {
                allocatedTam = futurTam;
                indexAux = (int*) realloc(*index, allocatedTam*sizeof(int));
                if (indexAux == NULL)
                    abort();
                *index = indexAux;
                futurTam = futurTam*2;
            }
            (*index)[count] = i;
            count++;
        }
    }
}
```



```

        return count;
    }

/**
 * @brief Set the value of Expected Positive Exposure, Expected Negative Exposure
 *        and Mark To Market Forward vectors in a given time step.
 *
 * @param [in] CashFlows Data.
 * @param [in] ExerciseTime Exercise dates information at time step 'step'.
 * @param [in] step Time step.
 * @param [in, out] EPE Expected Positive Exposure vector.
 * @param [in, out] ENE Expected Negative Exposure vector.
 * @param [in, out] MtMFwd Mark To Market Forward vector.
 */
static void getExposures(const PnlVect* CashFlows, const PnlVect* ExerciseTime,
                        int step, PnlVect* EPE, PnlVect* ENE, PnlVect* MtMFwd)
{
    if ((CashFlows == NULL) || (ExerciseTime == NULL) || (EPE == NULL) ||
        (ENE == NULL) || (MtMFwd == NULL))
        return;

    PnlVect* AuxPE = pnl_vect_copy(CashFlows);
    PnlVect* AuxNE = pnl_vect_copy(CashFlows);
    PnlVect* AuxMtMFwd = pnl_vect_copy(CashFlows);

    int i;
    for (i = 0; i < CashFlows->size; i++)
    {
        if (step > GET(ExerciseTime, i))
        {
            LET(AuxPE, i) = 0.;
            LET(AuxNE, i) = 0.;
        }
        else
        {
            LET(AuxPE, i) = GET(AuxPE, i) * exp(-r*(Dates[step]-Dates[0]));
            LET(AuxNE, i) = GET(AuxNE, i) * exp(-r*(Dates[step]-Dates[0]));
            LET(AuxMtMFwd, i) = GET(AuxMtMFwd, i) * exp(-r*(Dates[step]-Dates[0]));
            if (GET(CashFlows, i) < 0.)

```

```

        LET(AuxPE, i) = 0.;
    else
        LET(AuxNE, i) = 0.;
    }
}

LET(EPE, step) = pnl_vect_sum(AuxPE) / ((double) AuxPE->size);
LET(ENE, step) = pnl_vect_sum(AuxNE) / ((double) AuxNE->size);
LET(MtMFwd, step) = pnl_vect_sum(AuxMtMFwd) / ((double) AuxMtMFwd->size);

pnl_vect_free(&AuxPE);
pnl_vect_free(&AuxNE);
pnl_vect_free(&AuxMtMFwd);

return;
}

/**
 * @brief Generate the regression matrix of monomials.
 *
 * @param [in, out] RegrMat Regression matrix to fill.
 * @param [in] Nrows Number of rows of RegrMat.
 * @param [in] dataDim Dimension of regression matrix basis.
 * @param [in] maxDegree Maximum degree of regression matrix basis.
 * @param [in] XData Data with which generate matrix columns.
 */
static void regressMatMonos(PnlMat* RegrMat, int Nrows, int dataDim, int maxDegree,
                           const PnlMat* XData)
{
    assert(RegrMat != NULL);
    assert(XData != NULL);

    int i, j, k;
    double XDataValue;
    int basisValue;
    int maxDegreeAux;

    PnlBasis* basis;
    basis = pnl_basis_create_from_degree(PNL_BASIS_CANONICAL, maxDegree, dataDim

```

```

    pnl_mat_resize(RegrMat, Nrows, basis->T->m);

    for (i = 0; i < basis->T->m; i++)
    {
        for (k = 0; k < Nrows; k++)
        {
            XDataValue = 1.;
            maxDegreeAux = 0;
            for (j = 0; j < basis->T->n; j++)
            {
                basisValue = basis->T->array[i*(basis->T->n) + j];
                if (basisValue != 0)
                    XDataValue *= pow(MGET(XData, k, j), (double) basisValue);
                maxDegreeAux += basisValue;
                if (maxDegreeAux == maxDegree)
                    break;
            }
            MLET(RegrMat, k, i) = XDataValue;
        }
    }

    pnl_basis_free(&basis);

    return;
}

/**
 * @brief Generate the regression matrix for geometric basket options.
 *
 * @param [in, out] RegrMat Regression matrix to fill.
 * @param [in] Nrows Number of rows of RegrMat.
 * @param [in] dataDim Dimension of regression matrix basis.
 * @param [in] maxDegree Maximum degree of regression matrix basis.
 * @param [in] XData Data with which generate matrix columns.
 */
static void regressMatGBO(PnlMat* RegrMat, int Nrows, int dataDim, int maxDegree,
                          const PnlMat* XData)
{
    assert(RegrMat != NULL);
    assert(XData != NULL);

```

```

    int i, j;
    double XDataValue;

    pnl_mat_resize(RegrMat, Nrows, maxDegree+1);
    PnlVect* auxRow = pnl_vect_create(dataDim);

    for (i = 0; i < Nrows; i++)
    {
        pnl_mat_get_row(auxRow, XData, i);
        XDataValue = pow(pnl_vect_prod(auxRow), 1./((double) dataDim));
        for (j = 0; j <= maxDegree; j++)
        {
            if (j == 0)
                MLET(RegrMat, i, j) = 1.;
            else
                MLET(RegrMat, i, j) = pow(XDataValue, (double) j);
        }
    }

    pnl_vect_free(&auxRow);
    return;
}

/**
 * @brief Generate the regression matrix for arithmetic basket options.
 *
 * @param [in, out] RegrMat Regression matrix to fill.
 * @param [in] Nrows Number of rows of RegrMat.
 * @param [in] dataDim Dimension of regression matrix basis.
 * @param [in] maxDegree Maximum degree of regression matrix basis.
 * @param [in] XData Data with which generate matrix columns.
 */
static void regressMatABO(PnlMat* RegrMat, int Nrows, int dataDim, int maxDegree,
                        const PnlMat* XData)
{
    assert(RegrMat != NULL);
    assert(XData != NULL);

    int i, j;

```

```

double XDataValue;

pnl_mat_resize(RegrMat, Nrows, maxDegree+1);
PnlVect* auxRow = pnl_vect_create(dataDim);

for (i = 0; i < Nrows; i++)
{
    pnl_mat_get_row(auxRow, XData, i);
    XDataValue = pnl_vect_sum(auxRow) / ((double) dataDim);
    for (j = 0; j <= maxDegree; j++)
    {
        if (j == 0)
            MLET(RegrMat, i, j) = 1.;
        else
            MLET(RegrMat, i, j) = pow(XDataValue, (double) j);
    }
}

pnl_vect_free(&auxRow);
return;
}

static void (*getRegressionMatrix) (PnlMat*, int, int, int, const PnlMat*) = reg

/**
 * @brief Compute calibration of a model.
 *
 * @param [in] SPaths Simulated stochastic paths.
 * @param [out] a Coefficients found in each time step for each cluster.
 * @param [out] KMctrs Centres for k-means clustering algorithm for each time
 *                step. (numDates-2) matrices of Kxdim.
 * @param [out] ExerciseTime Early-exercise policy for each path.
 * @param [in] rng Random generator.
 *
 * @retval OK Calibration done correctly.
 * @retval ERR An error occurred.
 */
static int calibration(PnlMat** SPaths, PnlMat** a, PnlMat** KMctrs,
                      PnlVect** ExerciseTime, PnlRng* rng)

```

```

{
    assert(rng != NULL);

    if ( (SPaths == NULL) || (a == NULL) )
        return ERR;
    // There must be allocated memory for centroids in k-means
    if ( (ClustMethod == KMEANS) && (KMctrs == NULL) )
        return ERR;
    // Approximation dimension must be greater than 0
    if (H <= 0)
        return ERR;

    int* SPaths_Clusterized = (int*) malloc(NRepl*sizeof(int));
    if (SPaths_Clusterized == NULL)
        return ERR;
    int* ExerciseIdx = NULL;
    int* bundlePaths = NULL;
    // First set exercise time at expiration for convenience
    *ExerciseTime = pnl_vect_create_from_scalar(NRepl, numDates-1);

    // STEP II : Option value at terminal time
    PnlVect* CashFlows = pnl_vect_create(NRepl);
    getExerciseValue(SPaths, numDates-1, CashFlows);

    int i, j;
    int cluster;
    int step;
    int nbundlePaths; // number of paths in a bundle
    int nExerciseIdx; // number of Exercise Index
    PnlMat* RegrMat = pnl_mat_new();
    PnlMat* XData = pnl_mat_new();
    PnlMat* KMctrs_i = NULL;
    PnlVect* ContinuationValue = pnl_vect_create(NRepl);
    PnlVect* ExerciseValue = pnl_vect_create(NRepl);
    PnlVect* auxCashFlows = pnl_vect_new();
    PnlVect* auxCashFlows2;

    for (step = numDates-2; step > 0; step--)
    {
        // update the price
        getContinuationValue(SPaths, step, CashFlows, step+1, Dates);
    }
}

```

```

// STEP III : Bundling
// we do the clustering of the paths with the indicated method
if (clustering(SPaths[step], SPaths_Clusterized, &KMctrs_i, rng) == ERR)
{
    pnl_mat_free(&RegrMat);
    pnl_mat_free(&XData);
    pnl_vect_free(&ContinuationValue);
    pnl_vect_free(&ExerciseValue);
    pnl_vect_free(&auxCashFlows);
    pnl_vect_free(&CashFlows);
    free(SPaths_Clusterized);
    free(ExerciseIdx);
    free(bundlePaths);
    if (ClustMethod == KMEANS) pnl_mat_free(&KMctrs_i);
    return ERR;
}
if (ClustMethod == KMEANS)
{
    pnl_mat_clone(KMctrs[step-1], KMctrs_i);
    pnl_mat_free(&KMctrs_i);
}

for (cluster = 1; cluster <= K; cluster++)
{
    // STEP IV : Mapping high-dimesional state space to a
    //               low-dimensional space
    // We keep only the index of the paths of pertinent cluster
    nbundlePaths = findIndex_EqualValueInt(&bundlePaths, cluster,
                                           SPaths_Clusterized, NRepl);
    pnl_mat_resize(XData, nbundlePaths, dim);
    pnl_vect_resize(auxCashFlows, nbundlePaths);

    for (i = 0; i < nbundlePaths; i++)
    {
        for (j = 0; j < dim; j++)
        {
            MLET(XData, i, j) = MGET(SPaths[step], bundlePaths[i], j);
        }
        LET(auxCashFlows, i) = GET(CashFlows, bundlePaths[i]);
    }
}

```

```

//Begin computing continuation value
getRegressionMatrix(RegrMat, nbundlePaths, dim, H-1, XData);

// a sont les betas et RegrMat les phi du papier
if (pnl_mat_ls(RegrMat, auxCashFlows) == FAIL)
    abort();
if ((step == numDates-2) && (dim > 1))
{
    pnl_mat_resize(a[cluster-1], RegrMat->n, numDates-2);
}
pnl_mat_set_col(a[cluster-1], auxCashFlows, step-1);

// STEP V : Computing the continuation and option values at  $t_{m-1}$ 
// this is the Conditional expectation
auxCashFlows2 = pnl_mat_mult_vect(RegrMat, auxCashFlows);
for (i = 0; i < nbundlePaths; i++)
{
    LET(ContinuationValue, bundlePaths[i]) = GET(auxCashFlows2, i);
}
pnl_vect_free(&auxCashFlows2);
//End computing continuation value
}

// determining the exercise policy, not necessary for european options e
if (ExerciseType == AMERICAN)
{
    getExerciseValue(SPaths, step, ExerciseValue);

    // continuity condition to check at each step
    // we keep the index for which the continuity condition is satisfied
    nExerciseIdx = findIndex_GreaterValue(&ExerciseIdx, ExerciseValue,
                                         ContinuationValue);

    for (i = 0; i < nExerciseIdx; i++)
    {
        LET(CashFlows, ExerciseIdx[i]) = GET(ExerciseValue, ExerciseIdx[i])
        // tau* the optimal stopping time
        LET(*ExerciseTime, ExerciseIdx[i]) = step;
    }
}

```



```

    }

    pnl_mat_free(&RegrMat);
    pnl_mat_free(&XData);
    pnl_vect_free(&ContinuationValue);
    pnl_vect_free(&ExerciseValue);
    pnl_vect_free(&auxCashFlows);
    pnl_vect_free(&CashFlows);
    free(SPaths_Clusterized);
    free(ExerciseIdx);
    free(bundlePaths);

    return OK;
}

/**
 * @brief Compute the direct estimator.
 *
 * @param [in] SPaths Simulated stochastic paths.
 * @param [in] a Regression parameters/coefficients (coordonates in the basis).
 *           H coefficients in each time step for each cluster.
 * @param [in] KMctrs Centres for k-means clustering algorithm for each time
 *           step. (numDates-2) matrices of Kxdim.
 * @param [in] ExerciseTime Early-exercise policy for each path.
 * @param [out] directEstimator Computed direct estimator.
 * @param [in] rng Random generator.
 *
 * @retval OK Direct estimator computed correctly.
 * @retval ERR An error occurred.
 */
static int getDirectEstimator(PnlMat** SPaths, PnlMat** a, PnlMat** KMctrs,
                             PnlVect* ExerciseTime, double* directEstimator,
                             PnlRng* rng)
{
    assert(rng != NULL);

    if ((SPaths == NULL) || (a == NULL) || (ExerciseTime == NULL) ||
        (directEstimator == NULL))
        return ERR;
    // There must be centroids for k-means

```

```

if ( (ClustMethod == KMEANS) && (KMctrs == NULL) )
    return ERR;
// Approximation dimension must be greater than 0
if (H <= 0)
    return ERR;

int step = 1;
int* SPaths_Clusterized = (int*) malloc(NRepl*sizeof(int));
if (SPaths_Clusterized == NULL)
    return ERR;
PnlMat* KMctrs_i = NULL;
if (ClustMethod == KMEANS)
    KMctrs_i = pnl_mat_copy(KMctrs[step-1]);
// STEP III : Bundling, clustering of the paths with the indicated method
if (clustering(SPaths[step], SPaths_Clusterized, &KMctrs_i, rng) == ERR)
{
    free(SPaths_Clusterized);
    if (ClustMethod == KMEANS) pnl_mat_free(&KMctrs_i);
    return ERR;
}

int i, j;
int cluster;
int nbundlePaths; // number of paths in a bundle
int* bundlePaths = NULL;
PnlMat* XData = pnl_mat_new();
PnlMat* RegrMat = pnl_mat_new();
PnlVect* ContinuationValue = pnl_vect_create(NRepl);
PnlVect* auxContValue;
PnlVect* auxCoeff = pnl_vect_create(H);

for (cluster = 1; cluster <= K; cluster++)
{
    // STEP IV : Mapping high-dimesional state space to a low-dimensional sp
    // We keep only the index of the paths of pertinent cluster
    nbundlePaths = findIndex_EqualValueInt(&bundlePaths, cluster,
                                           SPaths_Clusterized, NRepl);
    pnl_mat_resize(XData, nbundlePaths, dim);
    for (i = 0; i < nbundlePaths; i++)
    {
        for (j = 0; j < dim; j++)

```

```

        {
            MLET(XData, i, j) = MGET(SPaths[step], bundlePaths[i], j);
        }
    }

//Begin computing continuation value
    getRegressionMatrix(RegrMat, nbundlePaths, dim, H-1, XData);

    // STEP V : Computing the continuation and option values at t_{m-1}
    // Conditional expectation
    pnl_mat_get_col(auxCoeff, a[cluster-1], step-1);
    auxContValue = pnl_mat_mult_vect(RegrMat, auxCoeff);
    for (i = 0; i < nbundlePaths; i++)
    {
        LET(ContinuationValue, bundlePaths[i]) = GET(auxContValue, i);
    }
    pnl_vect_free(&auxContValue);
//End computing continuation value
}

getContinuationValue(SPaths, step-1, ContinuationValue, step, Dates);
double ContValueMean = pnl_vect_sum(ContinuationValue) /
    ((double) ContinuationValue->size);

PnlVect* initialPayoff = pnl_vect_create(NRepl);
getExerciseValue(SPaths, step-1, initialPayoff);

*directEstimator = MAX(ContValueMean, GET(initialPayoff, 0));

pnl_mat_free(&RegrMat);
pnl_mat_free(&XData);
if (SPaths_Clusterized != NULL)
{
    free(SPaths_Clusterized);
    SPaths_Clusterized = NULL;
}
if (bundlePaths != NULL)
{
    free(bundlePaths);
    bundlePaths = NULL;
}

```

```

    pnl_vect_free(&initialPayoff);
    pnl_vect_free(&ContinuationValue);
    pnl_vect_free(&auxCoeff);
    if (ClustMethod == KMEANS) pnl_mat_free(&KMctrs_i);

    return OK;
}

/**
 * @brief Compute the path estimator.
 *
 * @param [in] SPaths Simulated stochastic paths.
 * @param [in] a Regression parameters/coefficient (coordinates in the basis).
 *              H coefficients in each time step for each cluster.
 * @param [in] KMctrs Centres for k-means clustering algorithm for each time
 *              step. (numDates-2) matrices of Kxdim.
 * @param [out] pathEstimator Computed path estimator.
 * @param [out] ExerciseTime Early-exercise policy for each path.
 * @param [in, out] EPE Expected Positive Exposure vector.
 * @param [in, out] ENE Expected Negative Exposure vector.
 * @param [in, out] MtMFwd Mark To Market Forward vector.
 * @param [in] rng Random generator.
 *
 * @retval OK Path estimator computed correctly.
 * @retval ERR An error occurred.
 */
static int getPathEstimator(PnlMat** SPaths, PnlMat** a, PnlMat** KMctrs,
                           double* pathEstimator, PnlVect** ExerciseTime,
                           PnlVect* EPE, PnlVect* ENE, PnlVect* MtMFwd, PnlRng*
{
    assert(rng != NULL);

    if ((SPaths == NULL) || (a == NULL) || (pathEstimator == NULL) ||
        (EPE == NULL) || (ENE == NULL) || (MtMFwd == NULL))
        return ERR;
    // There must be centroids for k-means
    if ( (ClustMethod == KMEANS) && (KMctrs == NULL) )
        return ERR;
    // Approximation dimension must be greater than 0
    if (H <= 0)

```

```

    return ERR;

    int* SPaths_Clusterized = (int*) malloc(NRepl*sizeof(int));
    if (SPaths_Clusterized == NULL)
        return ERR;
    int* ExerciseIdx = NULL;
    int* bundlePaths = NULL;
    // First set exercise time at expiration for convenience
    *ExerciseTime = pnl_vect_create_from_scalar(NRepl, numDates-1);

    // STEP II : Option value at terminal time
    PnlVect* CashFlows = pnl_vect_create(NRepl);
    getExerciseValue(SPaths, numDates-1, CashFlows);

    int i, j;
    int cluster;
    int step;
    int nbundlePaths; // number of paths in a bundle
    int nExerciseIdx; // number of Exercise Index
    PnlMat* RegrMat = pnl_mat_new();
    PnlMat* XData = pnl_mat_new();
    PnlMat* KMctrs_i = NULL;
    PnlVect* ContinuationValue = pnl_vect_create(NRepl);
    PnlVect* ExerciseValue = pnl_vect_create(NRepl);
    PnlVect* auxContValue;
    PnlVect* auxCoeff = pnl_vect_create(H);
    getExposures(CashFlows, *ExerciseTime, numDates-1, EPE, ENE, MtMFwd);

    for (step = numDates-2; step > 0; step--)
    {
        // update the price
        getContinuationValue(SPaths, step, CashFlows, step+1, Dates);

        // STEP III : Bundling
        if (ClustMethod == KMEANS)
            KMctrs_i = pnl_mat_copy(KMctrs[step-1]);
        if (clustering(SPaths[step], SPaths_Clusterized, &KMctrs_i, rng) == ERR)
        {
            free(SPaths_Clusterized);
            free(ExerciseIdx);
            free(bundlePaths);

```

```

    pnl_mat_free(&RegrMat);
    pnl_mat_free(&XData);
    pnl_vect_free(&CashFlows);
    pnl_vect_free(&ContinuationValue);
    pnl_vect_free(&ExerciseValue);
    pnl_vect_free(&auxCoeff);
    if (ClustMethod == KMEANS) pnl_mat_free(&KMctrs_i);
    return ERR;
}
if (ClustMethod == KMEANS) pnl_mat_free(&KMctrs_i);

for (cluster = 1; cluster <= K; cluster++)
{
    // STEP IV : Mapping high-dimesional state space to a
    //             low-dimensional space
    // We keep only the index of the paths of pertinent cluster
    nbundlePaths = findIndex_EqualValueInt(&bundlePaths, cluster,
                                           SPaths_Clusterized, NRepl);
    pnl_mat_resize(XData, nbundlePaths, dim);
    for (i = 0; i < nbundlePaths; i++)
    {
        for (j = 0; j < dim; j++)
        {
            MLET(XData, i, j) = MGET(SPaths[step], bundlePaths[i], j);
        }
    }

    //Begin computing continuation value
    getRegressionMatrix(RegrMat, nbundlePaths, dim, H-1, XData);

    // STEP V : Computing the continuation and option values at  $t_{m-1}$ 
    // Conditional expectation
    pnl_mat_get_col(auxCoeff, a[cluster-1], step-1);
    auxContValue = pnl_mat_mult_vect(RegrMat, auxCoeff);
    for (i = 0; i < nbundlePaths; i++)
    {
        LET(ContinuationValue, bundlePaths[i]) = GET(auxContValue, i);
    }
    pnl_vect_free(&auxContValue);
    //End computing continuation value
}

```

```

// determining exercise policy, not necessary for european options for e
if (ExerciseType == AMERICAN)
{
    getExerciseValue(SPaths, step, ExerciseValue);

    // continuity condition to check at each step
    // we keep the index for which the continuity condition is satisfied
    nExerciseIdx = findIndex_GreaterValue(&ExerciseIdx, ExerciseValue,
                                          ContinuationValue);

    for (i = 0; i < nExerciseIdx; i++)
    {
        LET(CashFlows, ExerciseIdx[i]) = GET(ExerciseValue, ExerciseIdx[i])
        // tau* the optimal stopping time
        LET(*ExerciseTime, ExerciseIdx[i]) = step;
    }
}

getExposures(CashFlows, *ExerciseTime, step, EPE, ENE, MtMFwd);
}
step++;
getContinuationValue(SPaths, step-1, CashFlows, step, Dates);

getExposures(CashFlows, *ExerciseTime, step-1, EPE, ENE, MtMFwd);

*pathEstimator = pnl_vect_sum(CashFlows) / ((double) CashFlows->size);

if (SPaths_Clusterized != NULL)
{
    free(SPaths_Clusterized);
    SPaths_Clusterized = NULL;
}
if (ExerciseIdx != NULL)
{
    free(ExerciseIdx);
    ExerciseIdx = NULL;
}
if (bundlePaths != NULL)
{
    free(bundlePaths);
}

```

```

        bundlePaths = NULL;
    }
    pnl_mat_free(&RegrMat);
    pnl_mat_free(&XData);
    pnl_vect_free(&CashFlows);
    pnl_vect_free(&ContinuationValue);
    pnl_vect_free(&ExerciseValue);
    pnl_vect_free(&auxCoeff);

    return OK;
}

```

```

int MC_JainOosterlee1ND(NumFunc_nd *p,PnlVect *s_0, PnlVect *volatility, double
{

    // Dimension
    dim = s_0->size;
    if (dim == 1)
        InstrumentType = SINGLE;
    else
        InstrumentType = BASKET;
//Option Type
    if ((p->Compute) == &CallBasket_nd)
    {
        OptionType = CALL;
        RegressMatType=ARITHMETIC;
    }
    else if ((p->Compute) == &PutBasket_nd)
    {
        OptionType = PUT;
        RegressMatType=ARITHMETIC;
    }
    else if ((p->Compute) == &CallGeom_nd)
    {
        OptionType = CALL;
        RegressMatType=GEOMETRIC;
    }
    else
    {
        OptionType = PUT;
    }
}

```



```

RegressMatType=GEOMETRIC;
}

// Spot
S0 = (double*) malloc(dim*sizeof(double));
if (S0 == NULL)
    return ERR;
memcpy(S0, s_0->array, dim*sizeof(double));
// Instantaneous Interest Rate
r = iir;
// Volatility
sigma = (double*) malloc(dim*sizeof(double));
if (sigma == NULL)
    return ERR;
memcpy(sigma, volatility->array, dim*sizeof(double));
// Continuous Dividend Rate
CDividendR = (double*) malloc(dim*sizeof(double));
if (CDividendR == NULL)
    return ERR;
memcpy(CDividendR, cdr->array, dim*sizeof(double));
// Correlation
if (InstrumentType == BASKET)
{
    if ( (corr <= 1) && (corr >= -1) && (corr >= (-1 / (dim-1))) )
        rho = corr;
    else
    {
        printf("\nIt is not a correct correlation value\n\n");
        return ERR;
    }
    setRhoMat();
    setSigmaMat();
}
// Maturity
T = maturity;
// Strike
Strike = strike;
// Monte Carlo simulations to perform
NRepl = NRC;
// Discretization steps to perform
Nsteps = NumSteps;
NumExerciseDates = 0;

```

```

ExerciseType = AMERICAN; // EUROPEAN AMERICAN

if (RegressMatType == GEOMETRIC)
{
    getRegressionMatrix = regressMatGB0;
    mappingF_ptr = geometricMean;
}
else if (RegressMatType == ARITHMETIC)
{
    getRegressionMatrix = regressMatAB0;
    mappingF_ptr = arithmeticMean;
}
else
{
    getRegressionMatrix = regressMatMonos;
    mappingF_ptr = arithmeticMean;
}
ClustMethod = clusteringMethod;
// approximation dimension
H = appDim;
// number of clusters
if (ClustMethod == KMEANS)
{
    K = numClusters;
}
else if (ClustMethod == RB)
{
    // 2^(Ncols*p) number of clusters for recursive bifurcation
    int p1 = (int) (log(numClusters)/(log(2.)*dim));
    K = (int) pow(2., p1*dim);
}
else if (ClustMethod == RBRSS)
{
    // 2^p number of clusters for recursive bifurcation RSS
    int p2 = (int) (log(numClusters)/log(2.));
    K = (int) pow(2., p2);
}
else // It is not a valid clustering method
    return ERR;

PnlRng* generator;

```

```

time_t tmp = time(NULL);
if (generatorType == PNL_RNG_MERSENNE_RANDOM_SEED)
{
    generator = pnl_rng_create(PNL_RNG_MERSENNE);
    pnl_rng_sseed(generator, tmp);
}
else
{
    generator = pnl_rng_create(generatorType);
    pnl_rng_sseed(generator, 0);
}

int i;
// vector with all dates creation
setDates();
PnlMat** SPaths = (PnlMat**) malloc(numDates*sizeof(PnlMat*));
if (SPaths == NULL)
    return ERR;
for (i = 0; i < numDates; i++)
    SPaths[i] = pnl_mat_new();
// paths generation
getTrajectories(SPaths, generator);

// memory allocation for regression coefficients and centers for kmeans
PnlMat** a = (PnlMat**) malloc(K*sizeof(PnlMat*));
if (a == NULL)
    return ERR;
for (i = 0; i < K; i++)
    a[i] = pnl_mat_create(H, numDates-2);
PnlMat** KMctrs = NULL;
if (ClustMethod == KMEANS)
{
    KMctrs = (PnlMat**) malloc((numDates-2)*sizeof(PnlMat*));
    if (KMctrs == NULL)
        return ERR;
    for (i = 0; i < numDates-2; i++)
        KMctrs[i] = pnl_mat_create(K, dim);
}
PnlVect* ExerciseTime = NULL;
PnlVect* ExerciseTime_PE = NULL;
PnlVect* EPE_PE = pnl_vect_create(numDates);

```

```

PnlVect* ENE_PE      = pnl_vect_create(numDates);
PnlVect* MtMFwd_PE = pnl_vect_create(numDates);

//BEGIN METHOD
if (calibration(SPaths, a, KMctrs, &ExerciseTime, generator) == ERR)
    return ERR;
if (getDirectEstimator(SPaths, a, KMctrs, ExerciseTime, directEstimator,
                      generator) == ERR)
    return ERR;
NRepl = NR;
if (generatorType == PNL_RNG_MERSENNE_RANDOM_SEED)
    pnl_rng_sseed(generator, tmp);
else
    pnl_rng_sseed(generator, 0);
getTrajectories(SPaths, generator);
if (getPathEstimator(SPaths, a, KMctrs, pathEstimator, &ExerciseTime_PE,
                    EPE_PE, ENE_PE, MtMFwd_PE, generator) == ERR)
    return ERR;
//END METHOD

pnl_rng_free(&generator);
if (ExerciseTime != NULL)
    pnl_vect_free(&ExerciseTime);
if (ExerciseTime_PE != NULL)
    pnl_vect_free(&ExerciseTime_PE);
pnl_vect_free(&EPE_PE);
pnl_vect_free(&ENE_PE);
pnl_vect_free(&MtMFwd_PE);
for (i = 0; i < numDates; i++)
{
    pnl_mat_free(&SPaths[i]);
}
free(SPaths);
SPaths = NULL;
for (i = 0; i < K; i++)
    pnl_mat_free(&a[i]);
free(a);
a = NULL;
if (ClustMethod == KMEANS)
{

```

```

        for (i = 0; i < numDates-2; i++)
        {
            pnl_mat_free(&KMctrs[i]);
        }
        free(KMctrs);
        KMctrs = NULL;
    }
    if (ExerciseDates != NULL)
    {
        free(ExerciseDates);
        ExerciseDates = NULL;
    }
    if (DiscretDates != NULL)
    {
        free(DiscretDates);
        DiscretDates = NULL;
    }
    if (Dates != NULL)
    {
        free(Dates);
        Dates = NULL;
    }
    if (S0 != NULL)
    {
        free(S0);
        S0 = NULL;
    }
    if (sigma != NULL)
    {
        free(sigma);
        sigma = NULL;
    }
    if (CDividendR != NULL)
    {
        free(CDividendR);
        CDividendR = NULL;
    }
    if (InstrumentType == BASKET)
        freeRhoSigmaMats();

return OK;

```

```
}
```

```
extern "C" {  
int CALC(MC_JainOosterlee1ND)(void *Opt, void *Mod, PricingMethod *Met)  
{  
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;  
    TYPEMOD *ptMod = (TYPEMOD *)Mod;  
    double r;  
    int i;  
    PnlVect *divid = pnl_vect_create(ptMod->Size.Val.V_PINT);  
    PnlVect *spot, *sig;  
  
    spot = ptMod->S0.Val.V_PNLVECT;  
    sig = ptMod->Sigma.Val.V_PNLVECT;  
    for (i = 0; i < ptMod->Size.Val.V_PINT; i++)  
        pnl_vect_set(divid, i, log(1. + GET(ptMod->Divid.Val.V_PNLVECT, i) / 100.));  
    r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);  
  
    MC_JainOosterlee1ND  
    (ptOpt->PayOff.Val.V_NUMFUNC_ND, spot,  
     sig,  
     ptMod->Rho.Val.V_RGDOUBLE,  
     ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,  
     ptOpt->PayOff.Val.V_NUMFUNC_2->Par[0].Val.V_PDOUBLE,  
     divid,  
     r,  
     Met->Par[0].Val.V_INT,  
     Met->Par[1].Val.V_LONG,  
     Met->Par[2].Val.V_LONG,  
     Met->Par[3].Val.V_ENUM.value,  
     Met->Par[4].Val.V_INT,  
     Met->Par[5].Val.V_ENUM.value,  
     Met->Par[6].Val.V_INT,  
     &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE));  
  
    pnl_vect_free(&divid);  
    return OK;  
}
```

```
static int CHK_OPT(MC_JainOosterlee1ND)(void *Opt, void *Mod)
```

```

{
    Option *ptOpt = (Option *)Opt;
    if ((strcmp(ptOpt->Name, "CallBasketAmer_nd") == 0) || (strcmp(ptOpt->Name, "Put
return WRONG;
}
}
#endif //PremiaCurrentVersion

extern "C" {
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->HelpFilenameHint = "MC_JainOosterlee1_ND";
        Met->Par[0].Val.V_INT = 10;
        Met->Par[1].Val.V_LONG = 50000;
        Met->Par[2].Val.V_LONG = 200000;
        Met->Par[3].Val.V_ENUM.value = 0;
        Met->Par[3].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[4].Val.V_INT = 3;
        Met->Par[5].Val.V_ENUM.value = 1;
        Met->Par[5].Val.V_ENUM.members = &PremiaEnumclusteringMethod;
        Met->Par[6].Val.V_INT = 4;

    }
    return OK;
}

PricingMethod MET(MC_JainOosterlee1ND) =
{
    "MC_JainOosterlee1ND",
    { {"Number of Exercise Dates", INT, {100}, ALLOW}, {"Number Replication Calibra
        {"Random Generator", ENUM, {0}, ALLOW},
        {"Approximation order", INT, {100}, ALLOW},
{"Clustering Method", ENUM, {0}, ALLOW},
        {"Number of Clusters", INT, {100}, ALLOW},
        {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_JainOosterlee1ND),
    { {"Direct Estimator", DOUBLE, {100}, FORBID}, {"Path Estimator", DOUBLE, {100

```

```
    {" ", PREMIA_NULLTYPE, {0}, FORBID}  
  },  
  CHK_OPT(MC_JainOosterlee1ND),  
  CHK_mc,  
  MET(Init)  
};  
}
```