

Help

```
/* Longstaff & Schwartz algorithm, backward simulated brownian paths */
/* Andersen & Broadie algorithm for dual upper bound computation*/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

#include "bsnd_stdnd.h"
#include "pnl/pnl_basis.h"
#include "black.h"
#include "optype.h"
#include "enums.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_mathtools.h"

FILE *pipo = NULL;
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2009+2) //The "#els
static int CHK_OPT(MC_AndersenBroadieND)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_AndersenBroadieND)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

/*For Primal method*/
static double *FP = NULL, *Paths = NULL, *PathsN = NULL;
static double *Brownian_Bridge = NULL;
static PnlMat *M = NULL, *BasisL2 = NULL;
static PnlVect *AuxR = NULL, *Res = NULL, *VBase = NULL;

/*For Dual method*/
static double *PathsInternal = NULL, *PathsNInternal = NULL;
static double *MM = NULL, *PI = NULL, *L = NULL, *Auxopt = NULL, *Auxscal = NULL;
static double *Brownian_Paths = NULL, *Brownian_PathsInternal = NULL;
```

```

static double *DiscountFactor = NULL;
static PnlVect *LowerDelta = NULL, *Spot_Internal = NULL;
static double *dist_test = NULL;
static PnlVect *ZERO = NULL, *Comput_time = NULL;
static PnlMat *S = NULL, *SN = NULL;

static PnlBasis *Basis;

static int LoScB_Allocation(long AL_MonteCarlo_Iterations,
                           int AL_Basis_Dimension, int BS_Dimension, int OP_Exercise_Dates -
{
    if (FP == NULL) FP = malloc(AL_MonteCarlo_Iterations * sizeof(double));
    if (FP == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (Paths == NULL) Paths = malloc(AL_MonteCarlo_Iterations * BS_Dimension * sizeof(double));
    if (Paths == NULL) return MEMORY_ALLOCATION_FAILURE;

    /* only usefull for normalised L&S, suboptimal but ... */
    if (PathsN == NULL)
    {
        PathsN = malloc(AL_MonteCarlo_Iterations * BS_Dimension * sizeof(double));
    }
    if (PathsN == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (BasisL2 == NULL) BasisL2 = pnl_mat_create_from_double(OP_Exercise_Dates -
    if (BasisL2 == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (M == NULL) M = pnl_mat_create(AL_Basis_Dimension, AL_Basis_Dimension);
    if (M == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (Brownian_Bridge == NULL)
    {
        Brownian_Bridge = malloc(AL_MonteCarlo_Iterations * BS_Dimension * sizeof(double));
    }
    if (Brownian_Bridge == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (Res == NULL) Res = pnl_vect_create(AL_Basis_Dimension);
    if (Res == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (AuxR == NULL) AuxR = pnl_vect_create(AL_Basis_Dimension);

```

```
if (AuxR == NULL) return MEMORY_ALLOCATION_FAILURE;

if (VBase == NULL) VBase = pnl_vect_create(AL_Basis_Dimension);
if (VBase == NULL) return MEMORY_ALLOCATION_FAILURE;

if (DiscountFactor == NULL)
{
    DiscountFactor = malloc(OP_Exercise_Dates * sizeof(double));
}
if (DiscountFactor == NULL) return MEMORY_ALLOCATION_FAILURE;

return OK;
}

static void LoScB_Liberation()
{
    if (FP != NULL)
    {
        free(FP);
        FP = NULL;
    }
    if (Brownian_Bridge != NULL)
    {
        free(Brownian_Bridge);
        Brownian_Bridge = NULL;
    }
    if (Paths != NULL)
    {
        free(Paths);
        Paths = NULL;
    }
    if (PathsN != NULL)
    {
        free(PathsN);
        PathsN = NULL;
    }
    if (M != NULL)
    {
        pnl_mat_free(&M);
        M = NULL;
    }
}
```

```

    }
    if (Res != NULL)
    {
        pnl_vect_free(&Res);
        Res = NULL;
    }
    if (AuxR != NULL)
    {
        pnl_vect_free(&AuxR);
        AuxR = NULL;
    }
    if (VBase != NULL)
    {
        pnl_vect_free(&VBase);
        VBase = NULL;
    }
}

```

```

static int AnBrB_Allocation(long AL_MonteCarlo_Iterations, long AL_MonteCarlo_It
                           int AL_Basis_Dimension, int BS_Dimension, int OP_Exe
{
    if (Paths == NULL)
    {
        Paths = malloc(AL_MonteCarlo_Iterations * BS_Dimension * sizeof(double));
    }
    if (Paths == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (PathsN == NULL)
    {
        PathsN = malloc(AL_MonteCarlo_Iterations * BS_Dimension * sizeof(double));
    }
    if (PathsN == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (PathsInternal == NULL) PathsInternal = malloc(AL_MonteCarlo_Iterations_Int
    if (PathsInternal == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (PathsNInternal == NULL)
    {

```

```

    PathsNInternal = malloc(AL_MonteCarlo_Iterations_Internal * BS_Dimension *
    }
    if (PathsNInternal == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (Brownian_Paths == NULL)
    {
        Brownian_Paths = malloc(AL_MonteCarlo_Iterations * BS_Dimension * sizeof(d
    }
    if (Brownian_Paths == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (Brownian_PathsInternal == NULL)
    {
        Brownian_PathsInternal = malloc(AL_MonteCarlo_Iterations_Internal * BS_Dim
    }
    if (Brownian_PathsInternal == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (MM == NULL)
    {
        MM = calloc(AL_MonteCarlo_Iterations, sizeof(double));
    }
    if (MM == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (PI == NULL)
    {
        PI = calloc(AL_MonteCarlo_Iterations, sizeof(double));
    }
    if (PI == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (L == NULL)
    {
        L = calloc(AL_MonteCarlo_Iterations, sizeof(double));
    }
    if (L == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (Auxopt == NULL)
    {
        Auxopt = calloc(AL_MonteCarlo_Iterations, sizeof(double));
    }
    if (Auxopt == NULL) return MEMORY_ALLOCATION_FAILURE;

```

```

    if (Auxscal == NULL)
    {
        Auxscal = calloc(AL_MonteCarlo_Iterations, sizeof(double));
    }
    if (Auxscal == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (estimate_mean == NULL)
    {
        estimate_mean = calloc(AL_MonteCarlo_Iterations, sizeof(double));
    }
    if (estimate_mean == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (LowerDelta == NULL) LowerDelta = pnl_vect_create(BS_Dimension);
    if (LowerDelta == NULL) return MEMORY_ALLOCATION_FAILURE;

    if (Spot_Internal == NULL) Spot_Internal = pnl_vect_create(BS_Dimension);
    if (Spot_Internal == NULL) return MEMORY_ALLOCATION_FAILURE;

    return OK;
}

static void AnBrB_Liberation()
{
    if (Paths != NULL)
    {
        free(Paths);
        Paths = NULL;
    }
    if (PathsN != NULL)
    {
        free(PathsN);
        PathsN = NULL;
    }
    if (PathsInternal != NULL)
    {
        free(PathsInternal);
        PathsInternal = NULL;
    }
    if (PathsNInternal != NULL)
    {
        free(PathsNInternal);
    }
}

```

```
    PathsNInternal = NULL;
}
if (Brownian_Paths != NULL)
{
    free(Brownian_Paths);
    Brownian_Paths = NULL;
}
if (Brownian_PathsInternal != NULL)
{
    free(Brownian_PathsInternal);
    Brownian_PathsInternal = NULL;
}
if (MM != NULL)
{
    free(MM);
    MM = NULL;
}
if (PI != NULL)
{
    free(PI);
    PI = NULL;
}
if (L != NULL)
{
    free(L);
    L = NULL;
}
if (Auxopt != NULL)
{
    free(Auxopt);
    Auxopt = NULL;
}
if (Auxscal != NULL)
{
    free(Auxscal);
    Auxscal = NULL;
}
if (estimate_mean != NULL)
{
    free(estimate_mean);
    estimate_mean = NULL;
}
```

```
    }  
    if (DiscountFactor != NULL)  
    {  
        free(DiscountFactor);  
        DiscountFactor = NULL;  
    }  
    if (BasisL2 != NULL)  
    {  
        pnl_mat_free(&BasisL2);  
    }  
    if (LowerDelta != NULL)  
    {  
        pnl_vect_free(&LowerDelta);  
    }  
    if (Spot_Internal != NULL)  
    {  
        pnl_vect_free(&Spot_Internal);  
    }  
    if (dist_test != NULL)  
    {  
        free(dist_test);  
        dist_test = NULL;  
    }  
    if (ZERO != NULL)  
    {  
        pnl_vect_free(&ZERO);  
    }  
    if (Comput_time != NULL)  
    {  
        pnl_vect_free(&Comput_time);  
    }  
    if (S != NULL)  
    {  
        pnl_mat_free(&S);  
    }  
    if (SN != NULL)  
    {  
        pnl_mat_free(&SN);  
    }  
}
```



```

static void Regression(long AL_MonteCarlo_Iterations, NumFunc_nd *p,
                      int AL_Basis_Dimension, int BS_Dimension, int Time,
                      int AL_PayOff_As_Regressor, int use_normalised_regressor,
                      double step)
{
    int i, j, k;
    double AuxOption, tmp;
    double *PathspkmDimBS = Paths, *PathsNpkmDimBS = Paths;
    PnlVect VStock;
    long InTheMonney = 0;
    VStock.size = BS_Dimension;

    if (use_normalised_regressor)
        PathsNpkmDimBS = PathsN;

    pnl_vect_set_double(AuxR, 0.0);
    pnl_mat_set_double(M, 0.0);

    for (k = 0; k < AL_MonteCarlo_Iterations; k++)
    {
        /*kth regressor value*/
        VStock.array = &(PathspkmDimBS[k * BS_Dimension]);
        AuxOption = p->Compute(p->Par, &VStock);
        /*only the at-the-monney path are taken into account*/
        if (AuxOption > 0)
        {
            InTheMonney++;
            /*value of the regressor basis on the kth path*/
            if (AL_PayOff_As_Regressor == 1)
            {
                /*here, the payoff function is introduced in the regression basis*/
                LET(VBase, 0) = AuxOption;
                for (i = 1; i < AL_Basis_Dimension; i++)
                {
                    LET(VBase, i) = pnl_basis_i(Basis, &(PathsNpkmDimBS[k * BS_Di
                }
            }
        }
        else
        {

```

```

        for (i = 0; i < AL_Basis_Dimension; i++)
        {
            LET(VBase, i) = pnl_basis_i(Basis, &(PathsNpkmDimBS[k * BS_Di
        }
    }
    /*empirical regressor dispersion matrix*/
    for (i = 0; i < AL_Basis_Dimension; i++)
        for (j = 0; j < AL_Basis_Dimension; j++)
        {
            tmp = MGET(M, i, j);
            MLET(M, i, j) = tmp + GET(VBase, i) * GET(VBase, j);
        }
    /*auxiliary for regression formulae*/
    for (i = 0; i < AL_Basis_Dimension; i++)
    {
        tmp = GET(AuxR, i);
        LET(AuxR, i) = FP[k] * GET(VBase, i) + tmp;
    }
}

}
if (InTheMonney == 0)
{
    pnl_vect_set_double(Res, 0);
}
else
{
    pnl_vect_clone(Res, AuxR);
    pnl_mat_ls(M, Res);
}
}

/*Generates the Continuation value lower bounds thanks to Broadie/Cao article :
static double European_Lower_bound(PnlVect *BS_Spot,
                                   NumFunc_nd *p,
                                   double OP_Maturity,
                                   double BS_Interest_Rate,
                                   PnlVect *BS_Dividend_Rate,
                                   PnlVect *BS_Volatility,
                                   double *BS_Correlation,
                                   double Time)
{

```

```

int BS_Dimension = BS_Spot->size;
double lower_bound = 0.;
double Strike = p->Par[0].Val.V_DOUBLE;

/* if ((p->Compute)==&PutBasket_nd)
*   {
*       ap_carmonadurleman(BS_Spot,p,OP_Maturity-Time,BS_Interest_Rate,BS_Divi
*       BS_Volatility,BS_Correlation[BS_Dimension+1],&lower
*       return(lower_bound);
*   } */
if ((p->Compute) == &CallBasket_nd)
{
    lower_bound = European_call_put_geometric_mean(BS_Spot, Time, OP_Maturity,
        BS_Dimension, BS_Interest_Rate,
        BS_Dividend_Rate, BS_Volatility->array,
        BS_Correlation, TRUE);
    return (lower_bound);
}
if ((p->Compute) == &PutGeom_nd)
{
    lower_bound = European_call_put_geometric_mean(BS_Spot, Time, OP_Maturity,
        BS_Dimension, BS_Interest_Rate,
        BS_Dividend_Rate, BS_Volatility->array,
        BS_Correlation, FALSE);
    return (lower_bound);
}

if ((p->Compute) == &CallMax_nd)
{
    lower_bound = European_call_price_average(BS_Spot, Time, OP_Maturity, Stri
        BS_Interest_Rate, BS_Dividend_Rate);
    return (lower_bound);
}

return (0.);
}

/*see the documentation for the parameters meaning*/
int AnBrB(PnlVect *BS_Spot,
    NumFunc_nd *p,

```

```

        double OP_Maturity,
        double BS_Interest_Rate,
        PnlVect *BS_Dividend_Rate,
        PnlVect *BS_Volatility,
        double *BS_Correlation,
        long AL_MonteCarlo_Iterations_Primal,
        int generator,
        int name_basis,
        int AL_Basis_Dimension,
        int OP_Exercise_Dates,
        int AL_PayOff_As_Regressor,
        int AL_Antithetic,
        int use_normalised_regressor,
        long AL_MonteCarlo_Iterations_Dual,
        long AL_MonteCarlo_Iterations_Dual_Internal,
        int use_boundary_distance_grouping,
        double *Dual_Price)
{
    double AuxOption, AuxScal, DiscountStep, Step, reg, AuxOption_internal, AuxScal;
    int i;
    int k, l, init_mc, init, m;
    int BS_Dimension = BS_Spot->size;
    double *paths; /* = Paths but changed to PathsN, when use_normalised_regressor */
    PnlVect VStock;

    /*For Primal method*/
    double AL_FPrice = 0.;

    /*For Dual method*/
    double *pathsInternal; /* = PathsInternal but changed to PathsNInternal, when */

    /*For Boundary distance grouping*/
    //double dist=0.,E=0.;
    int nb_dist_test = 15;
    double dist = 0., E = 0., mu0 = 0., mu1 = 0., sig20 = 0., sig21 = 0., pz = 0.,
    double TP = 0., TI = 0., T0 = 0., T1 = 0.;
    /*TP : estimated time to generate a Path, TI: time to determine which group th
    T0: time to compute an increment in group ZERO, T1: time to compute an incre
    double initial_time, final_time;
    long AL_MonteCarlo_Iterations_Dual_prelim = AL_MonteCarlo_Iterations_Dual;
    long AL_MonteCarlo_Iterations_Dual_Internal_prelim = AL_MonteCarlo_Iterations_

```

```

double alpha_opt = 1.;
double opt_dist = 1000.0;
int lz, n0 = 0, n1 = 0;
double E0 = 0., E1 = 0.;

/*For Dual method*/
*Dual_Price = 0.;

/* MC sampling */
init_mc = pnl_rand_init(generator, BS_Dimension, AL_MonteCarlo_Iterations_Primal, BS_Dimension);

/* Test after initialization for the generator */
if (init_mc != OK) return init_mc;

/* initialisation of BS */
init = Init_BS(BS_Dimension, BS_Volatility->array, BS_Correlation,
               BS_Interest_Rate, BS_Dividend_Rate->array);
if (init != OK) return init;

/*Initialization of the regression basis*/
Basis = pnl_basis_create_from_degree(name_basis, AL_Basis_Dimension, BS_Dimension);

/*time step*/
Step = OP_Maturity / (double)(OP_Exercise_Dates - 1);
/*discounting factor for a time step*/
DiscountStep = exp(-BS_Interest_Rate * Step);

/*memory allocation of the algorithm's variables*/
init = LoScB_Allocation(AL_MonteCarlo_Iterations_Primal, AL_Basis_Dimension, BS_Dimension);
if (init != OK) return init;
paths = Paths;

if (AL_Antithetic)
    /*here, the brownian bridge is initialised with antithetic paths*/
    Init_Brownian_Bridge_A(Brownian_Bridge, AL_MonteCarlo_Iterations_Primal,
                           BS_Dimension, OP_Maturity, generator);
else
    Init_Brownian_Bridge(Brownian_Bridge, AL_MonteCarlo_Iterations_Primal,
                           BS_Dimension, OP_Maturity, generator);

/*computation of the BlackScholes paths at the maturity related to Brownian_Bridge*/

```

```

Backward_Path(Paths, Brownian_Bridge, BS_Spot->array, OP_Maturity,
              AL_MonteCarlo_Iterations_Primal, BS_Dimension);

/*initialisation of the payoff values at the maturity*/
for (i = 0; i < AL_MonteCarlo_Iterations_Primal; i++)
{
    VStock.size = BS_Dimension;
    VStock.array = &(Paths[i * BS_Dimension]);
    FP[i] = p->Compute(p->Par, &VStock);
    if (FP[i] > 0) FP[i] = DiscountStep * FP[i];
}

for (k = OP_Exercice_Dates - 2; k >= 1; k--)
{
    if (AL_Antithetic)
        /*here, the brownian bridge is computed with antithetic paths*/
        Compute_Brownian_Bridge_A(Brownian_Bridge, k * Step, Step, BS_Dimension,
                                   AL_MonteCarlo_Iterations_Primal, generator);
    else
        Compute_Brownian_Bridge(Brownian_Bridge, k * Step, Step, BS_Dimension,
                                   AL_MonteCarlo_Iterations_Primal, generator);

    /*computation of the BlackScholes paths at time k related to Brownian_Brid
    Backward_Path(Paths, Brownian_Bridge, BS_Spot->array, (double)k * Step,
                  AL_MonteCarlo_Iterations_Primal, BS_Dimension);

    if (use_normalised_regressor)
    {
        /*computation of the inverse of the BlackScholes
        * dispersion matrix used in the normalisation paths
        * procedure*/
        Compute_Inv_Sqrt_BS_Dispersion((double)k * Step, BS_Dimension, BS_Spot
                                       BS_Interest_Rate, BS_Dividend_Rate);

        /*the regression is done with respect to the normalised
        * BlackScholes paths (see the documentation)*/
        NormalisedPaths(Paths, PathsN, AL_MonteCarlo_Iterations_Primal, BS_Dim
        paths = PathsN;
    }

```

```

/*regression procedure*/
Regression(AL_MonteCarlo_Iterations_Primal, p, AL_Basis_Dimension,
          BS_Dimension, k, AL_PayOff_As_Regressor, use_normalised_regression
          Step);

/*regression factors kept in the L2 basis matrix*/
pnl_mat_set_row(BasisL2, Res, k);

/* dynamical programming*/
for (i = 0; i < AL_MonteCarlo_Iterations_Primal; i++)
{
    /*exercise value*/
    VStock.size = BS_Dimension;
    VStock.array = &(Paths[i * BS_Dimension]);
    AuxOption = p->Compute(p->Par, &VStock);
    /*approximated continuation value, only the at-the-money paths are taken*/
    if (AuxOption > 0)
    {
        /* if k is greater than or equal to
         * AL_PayOff_As_Regressor, the payoff function is
         * introduced to the regression basis*/
        if (AL_PayOff_As_Regressor == 1)
        {
            AuxScal = AuxOption * GET(Res, 0);
            for (l = 1; l < AL_Basis_Dimension; l++)
                AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimension, l - 1);
        }
        else
        {
            AuxScal = 0.;
            for (l = 0; l < AL_Basis_Dimension; l++)
            {
                AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimension, l);
            }
        }
        /* AuxScal contains the approximated continuation value*/
        /* if the continuation value is less than the exercise value,
         * the optimal stopping time is modified*/
        if (AuxOption > AuxScal)

```

```

        FP[i] = AuxOption;

    }
    /*Discount for a time step*/
    FP[i] *= DiscountStep;
}

}
/*at time 0, the conditionnal expectation reduces to an expectation*/
for (i = 0; i < AL_MonteCarlo_Iterations_Primal; i++)
{
    AL_FPrice += FP[i];
}

AL_FPrice /= (double)AL_MonteCarlo_Iterations_Primal;

/*Continuation value at time=0 is kept in the L2 Basis*/
MLET(BasisL2, 0, 0) = AL_FPrice;

/*output of the L&S algorithm*/
AL_FPrice = MAX(p->Compute(p->Par, BS_Spot), AL_FPrice);

/*memory liberation*/
LoScB_Liberation();

//Algorithm for Dual Method(Andersen/Broadie) based on the L&S regression
// to approximate option's continuation values

/*memory allocation of the algorithm's variables*/
init = AnBrB_Allocation(AL_MonteCarlo_Iterations_Dual, AL_MonteCarlo_Iteration
                        AL_Basis_Dimension, BS_Dimension, OP_Exercice_Dates);
if (init != OK) return init;
paths = Paths;
pathsInternal = PathsInternal;

//Discount Factors vector

DiscountFactor[0] = 1;

for (k = 1; k < OP_Exercice_Dates; k++)

```


[illegible]

```

    NormalisedPaths(Paths, PathsN, AL_MonteCarlo_Iterations_Dual_prelim);
    paths = PathsN;
}

for (i = 0; i < AL_MonteCarlo_Iterations_Dual_prelim; i++)
{
    VStock.size = BS_Dimension;
    VStock.array = &(Paths[i * BS_Dimension]);
    AuxOption = p->Compute(p->Par, &VStock);

    if (k == OP_Exercise_Dates - 1)
    {
        AuxScal = 0.;
    }

    else
    {
        if (AL_PayOff_As_Regressor <= 1)
        {
            reg = MGET(BasisL2, k, 0);
            AuxScal = AuxOption * reg;
            for (l = 1; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimension, l);
            }
        }
        else
        {
            AuxScal = 0.;
            for (l = 0; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimension, l);
            }
        }
    }

    for (l = 0; l < BS_Dimension; l++)
    {

```

```

        LET(Spot_Internal, l) = Paths[i * BS_Dimension + l];
    }

    dist += fabs(MAX(European_Lower_bound(Spot_Internal, p, OP_Maturit
    }
}

dist /= (double)((OP_Exercice_Dates) * AL_MonteCarlo_Iterations_Dual_prelim);

//Set of possible test distances, built on the previous average
if (dist_test == NULL) dist_test = malloc(nb_dist_test * sizeof(double));
if (dist_test == NULL) return MEMORY_ALLOCATION_FAILURE;

for (l = 0; l < nb_dist_test; l++)
{
    dist_test[l] = dist * (nb_dist_test - 1) / (double)100;
}

//Preliminary step : estimation of optimal lZ and Delta for distance group

if (ZERO == NULL) ZERO = pnl_vect_create_from_double(AL_MonteCarlo_Iterati
if (ZERO == NULL) return MEMORY_ALLOCATION_FAILURE;

if (Comput_time == NULL) Comput_time = pnl_vect_create_from_double(AL_Mont
if (Comput_time == NULL) return MEMORY_ALLOCATION_FAILURE;

for (l = 0; l < nb_dist_test; l++)
{
    dist = dist_test[l];
    mu0 = 0., mu1 = 0., sig20 = 0., sig21 = 0., pz = 0.;

    // Martingale initialization according to Andersen/Broadie Algorithm

    //Initial Step : k=0

    for (i = 0; i < AL_MonteCarlo_Iterations_Dual_prelim; i++)
    {

```

```

LET(ZERO, i) = 0.;
LET(Comput_time, i) = 0.;

initial_time = clock();
AuxOption = p->Compute(p->Par, BS_Spot);
AuxScal = MGET(BasisL2, 0, 0);
L[i] = AL_FPrice;
Auxopt[i] = AuxOption;
Auxscal[i] = AuxScal;
PI[i] = AL_FPrice;
MM[i] = AuxOption * DiscountFactor[0] - PI[i];
final_time = clock();
LET(Comput_time, i) = (GET(Comput_time, i)) + (final_time - initial_time);

initial_time = clock();
E = European_Lower_bound(BS_Spot, p, OP_Maturity, BS_Interest_Rate, BS_Volatility, BS_Correlation);
dt = fabs(MAX(E, Auxscal[i]) - Auxopt[i]);

if (dt < dist && Auxopt[i] > E && Auxopt[i] > Auxscal[i]) LET(ZERO, i) = 1;
final_time = clock();
TI += (final_time - initial_time);
}

//Building the martingale thanks to Andersen/Broadie algorithm, from s
for (k = 1; k <= OP_Exercice_Dates - 1; k++)
{
  for (i = 0; i < AL_MonteCarlo_Iterations_Dual_prelim; i++)
  {
    initial_time = clock();

    for (l = 0; l < BS_Dimension; l++)
    {
      LET(Spot_Internal, l) = Paths[i * BS_Dimension + l];
    }
    estimate_mean[i] = 0.;
    if (Auxopt[i] > Auxscal[i] &&
        Auxopt[i] > European_Lower_bound(Spot_Internal, p, OP_Maturity, BS_Volatility, BS_Correlation))
    {

```

```

if (AL_Antithetic)
    /*here, the brownian path is computed with antithetic pa
    Compute_Brownian_Paths_A(Brownian_PathsInternal, sqrt((d
                                BS_Dimension, AL_MonteCarlo_Ite
                                generator);
else
    Compute_Brownian_Paths(Brownian_PathsInternal, sqrt((dou
                                BS_Dimension, AL_MonteCarlo_Itera
                                generator);

/*computation of the BlackScholes paths at time k related

BS_Forward_Path(PathsInternal, Brownian_PathsInternal, Spo
                (double) Step, AL_MonteCarlo_Iterations_Du

if (use_normalised_regressor)
{
    Compute_Inv_Sqrt_BS_Dispersion((double)Step, BS_Dimens
                                BS_Interest_Rate, BS_Di

    NormalisedPaths(PathsInternal, PathsNInternal, AL_Mont
    pathsInternal = PathsNInternal;
}

for (m = 0; m < AL_MonteCarlo_Iterations_Dual_Internal_pre
{
    VStock.size = BS_Dimension;
    VStock.array = &(PathsInternal[m * BS_Dimension]);
    AuxOption_internal = p->Compute(p->Par, &VStock);

    if (k == OP_Exercise_Dates - 1)
    {
        AuxScal_internal = 0.;
    }
    else
    {
        if (AL_PayOff_As_Regressor <= k)
        {
            reg = MGET(BasisL2, k, 0);

```

```

        AuxScal_internal = AuxOption_internal * reg;

        for (l = 1; l < AL_Basis_Dimension; l++)
        {
            reg = MGET(BasisL2, k, l);
            AuxScal_internal += pnl_basis_i(Basis, pat
        }
    }

    else
    {
        AuxScal_internal = 0.;
        for (l = 0; l < AL_Basis_Dimension; l++)
        {
            reg = MGET(BasisL2, k, l);
            AuxScal_internal += pnl_basis_i(Basis, pat
        }
    }

    estimate_mean[i] += MAX(AuxScal_internal, AuxOption_in
}

        final_time = clock();
        LET(Comput_time, i) = (GET(Comput_time, i)) + (final_time
    }

}

initial_time = clock();

if (AL_Antithetic)
    /*here, the brownian path is computed with antithetic paths*/
    Compute_Brownian_Paths_A(Brownian_Paths, sqrt((double)k * Step),
        BS_Dimension, AL_MonteCarlo_Iterations_Du
        generator);
else
    Compute_Brownian_Paths(Brownian_Paths, sqrt((double)k * Step),
        BS_Dimension, AL_MonteCarlo_Iterations_Du
        generator);

```

```

/*computation of the BlackScholes paths at time k related to Brown
BS_Forward_Path(Paths, Brownian_Paths, BS_Spot->array, k * Step, A

if (use_normalised_regressor)
{
    Compute_Inv_Sqrt_BS_Dispersion(k * Step, BS_Dimension, BS_Spot
                                   BS_Interest_Rate, BS_Dividend_R

    NormalisedPaths(Paths, PathsN, AL_MonteCarlo_Iterations_Dual_p
    paths = PathsN;
}

final_time = clock();
TP += (final_time - initial_time);

for (i = 0; i < AL_MonteCarlo_Iterations_Dual_prelim; i++)
{
    for (l = 0; l < BS_Dimension; l++)
    {
        LET(Spot_Internal, l) = Paths[i * BS_Dimension + l];
    }

    initial_time = clock();
    VStock.size = BS_Dimension;
    VStock.array = &(Paths[i * BS_Dimension]);
    AuxOption = p->Compute(p->Par, &VStock);

    if (k == OP_Exercice_Dates - 1)
    {
        AuxScal = 0.;
    }
    else
    {
        if (AL_PayOff_As_Regressor <= k)
        {
            reg = MGET(BasisL2, k, 0);
            AuxScal = AuxOption * reg;
            for (l = 1; l < AL_Basis_Dimension; l++)
            {

```

```

        reg = MGET(BasisL2, k, l);
        AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimen
    }
    }
else
{
    AuxScal = 0.;
    for (l = 0; l < AL_Basis_Dimension; l++)
    {
        reg = MGET(BasisL2, k, l);
        AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimen
    }
}
}
if (Auxopt[i] <= Auxscal[i])
{
    PI[i] += MAX(AuxOption, AuxScal) * DiscountFactor[k] - L[i]
}
else
{
    PI[i] += MAX(AuxOption, AuxScal) * DiscountFactor[k] - est
}
L[i] = MAX(AuxOption, AuxScal);
Auxopt[i] = AuxOption;
Auxscal[i] = AuxScal;
MM[i] = MAX(MM[i], AuxOption * DiscountFactor[k] - PI[i]);
final_time = clock();
LET(Comput_time, i) = (GET(Comput_time, i)) + (final_time - in

if (GET(ZERO, i) == 0)
{
    initial_time = clock();
    E = European_Lower_bound(Spot_Internal, p, OP_Maturity, BS
    dt = fabs(MAX(E, Auxscal[i]) - Auxopt[i]);
    if (dt < dist && Auxopt[i] > E && Auxopt[i] > Auxscal[i])
        final_time = clock();
    TI += (final_time - initial_time);
}
}
}

```



```

for (i = 0; i < AL_MonteCarlo_Iterations_Dual_prelim; i++)
{
    pz += (GET(ZERO, i)) / (double)AL_MonteCarlo_Iterations_Dual_prelim;
}

for (i = 0; i < AL_MonteCarlo_Iterations_Dual_prelim; i++)
{
    if (GET(ZERO, i) == 1)
    {
        mu1 += MM[i];
        sig21 += MM[i] * MM[i];
        T1 += GET(Comput_time, i);
    }
    if (GET(ZERO, i) == 0)
    {
        mu0 += MM[i];
        sig20 += MM[i] * MM[i];
        T0 += GET(Comput_time, i);
    }
}

mu1 /= (double)pz;
mu1 /= (double)AL_MonteCarlo_Iterations_Dual_prelim;

mu0 /= (double)(1 - pz);
mu0 /= (double)AL_MonteCarlo_Iterations_Dual_prelim;

sig21 /= (double)pz;
sig21 /= (double)AL_MonteCarlo_Iterations_Dual_prelim;
sig21 -= mu1 * mu1;

sig20 /= (double)(1 - pz);
sig20 /= (double)AL_MonteCarlo_Iterations_Dual_prelim;
sig20 -= mu0 * mu0;

```

```

TP /= (double)AL_MonteCarlo_Iterations_Dual_prelim;
TI /= (double)AL_MonteCarlo_Iterations_Dual_prelim;

T1 /= (double)pz;
T1 /= (double)AL_MonteCarlo_Iterations_Dual_prelim;

T0 /= (double)(1 - pz);
T0 /= (double)AL_MonteCarlo_Iterations_Dual_prelim;

alpha = sqrt((1 - pz) * (1 - pz) * sig20 * (TP + TI + pz * T1) / (pz *

//Find the optimal alpha : see documentation

if ((alpha < alpha_opt) && (alpha > 0.01))
{
    alpha_opt = alpha;
    opt_dist = dist;
}

if (alpha_opt < 0.2) break;
}

//Computation of the alternative estimator : see documentation
lz = floor(alpha_opt * AL_MonteCarlo_Iterations_Dual);

if (S == NULL) S = pnl_mat_create(OP_Exercise_Dates, AL_MonteCarlo_Iterati
if (S == NULL) return MEMORY_ALLOCATION_FAILURE;

if (use_normalised_regressor)
{
    if (SN == NULL) SN = pnl_mat_create(OP_Exercise_Dates, AL_MonteCarlo_I
    if (SN == NULL) return MEMORY_ALLOCATION_FAILURE;
}

pnl_vect_resize(ZERO, AL_MonteCarlo_Iterations_Dual);
pnl_vect_set_double(ZERO, 0.);

//Sampling all the paths and check the group that each path belongs to

```

```

//Step k=0

AuxOption = p->Compute(p->Par, BS_Spot);
AuxScal = MGET(BasisL2, 0, 0);
E = European_Lower_bound(BS_Spot, p, OP_Maturity, BS_Interest_Rate, BS_Div
dist = fabs(MAX(E, AuxScal) - AuxOption);

if ((dist < opt_dist) && (AuxOption > MAX(E, AuxScal)))
{
    pnl_vect_set_double(ZERO, 1.);
}

//Steps from 1 to OP_Exercise_Dates-1
for (k = 1; k < OP_Exercise_Dates; k++)
{
    if (AL_Antithetic)
        /*here, the brownian path is computed with antithetic paths*/
        Compute_Brownian_Paths_A(Brownian_Paths, sqrt((double)k * Step),
                                BS_Dimension, AL_MonteCarlo_Iterations_Dual,
                                generator);
    else
        Compute_Brownian_Paths(Brownian_Paths, sqrt((double)k * Step),
                                BS_Dimension, AL_MonteCarlo_Iterations_Dual,
                                generator);

    /*computation of the BlackScholes paths at time k related to Brownian
    BS_Forward_Path(Paths, Brownian_Paths, BS_Spot->array, (double) k * St

    if (use_normalised_regressor)
    {

        Compute_Inv_Sqrt_BS_Dispersion((double)k * Step, BS_Dimension, BS_
                                BS_Interest_Rate, BS_Dividend_Rate)

        NormalisedPaths(Paths, PathsN, AL_MonteCarlo_Iterations_Dual, BS_D
        paths = PathsN;

        for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
        {
            //Paths are kept in memory

```

```

        for (l = 0; l < BS_Dimension; l++)
        {
            MLET(SN, k, i * BS_Dimension + l) = PathsN[i * BS_Dimension + l]
        }
    }

//Paths are kept in memory
for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
{
    for (l = 0; l < BS_Dimension; l++)
    {
        MLET(S, k, i * BS_Dimension + l) = Paths[i * BS_Dimension + l]
    }
}

for (i = 0; i < AL_MonteCarlo_Iterations_Dual_prelim; i++)
{
    VStock.size = BS_Dimension;
    VStock.array = &(Paths[i * BS_Dimension]);
    AuxOption = p->Compute(p->Par, &VStock);

    if (k == OP_Exercice_Dates - 1)
    {
        AuxScal = 0.;
    }
    else
    {
        if (AL_PayOff_As_Regressor <= k)
        {
            reg = MGET(BasisL2, k, 0);
            AuxScal = AuxOption * reg;
            for (l = 1; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimension, l);
            }
        }
        else
        {
            AuxScal = 0.;
        }
    }
}

```

```

        for (l = 0; l < AL_Basis_Dimension; l++)
        {
            reg = MGET(BasisL2, k, l);
            AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimension
        }
    }

if (GET(ZERO, i) == 0)
{
    for (l = 0; l < BS_Dimension; l++)
    {
        LET(Spot_Internal, l) = Paths[i * BS_Dimension + l];
    }

    E = European_Lower_bound(Spot_Internal, p, OP_Maturity, BS_Int
        BS_Volatility, BS_Correlation, (doubl
    dist = fabs(MAX(E, AuxScal) - AuxOption);

    if ((dist < opt_dist) && (AuxOption > MAX(E, AuxScal))) LET(ZE
}
}

}

//Martingale Building according to Broadie/Cao Algorithm, based on Anderse

paths = Paths; //Reinitialize paths

//Initial Step : k=0
n0 = 0;
for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
{
    if ((GET(ZERO, i) == 1) || ((GET(ZERO, i) == 0) && (n0 < lz)))
    {
        if ((GET(ZERO, i) == 0) && (n0 < lz)) n0++;

        AuxOption = p->Compute(p->Par, BS_Spot);
        AuxScal = MGET(BasisL2, 0, 0);
        L[i] = AL_FPrice;
        Auxopt[i] = AuxOption;

```

[illegible]

```

/*computation of the BlackScholes paths at time k related
BS_Forward_Path(PathsInternal, Brownian_PathsInternal, Spo

if (use_normalised_regressor)
{
    Compute_Inv_Sqrt_BS_Dispersion((double)Step, BS_Dimens
                                BS_Interest_Rate, BS_Di

    NormalisedPaths(PathsInternal, PathsNInternal, AL_Mont
    pathsInternal = PathsNInternal;
}

for (m = 0; m < AL_MonteCarlo_Iterations_Dual_Internal; m++)
{
    VStock.size = BS_Dimension;
    VStock.array = &(PathsInternal[m * BS_Dimension]);
    AuxOption_internal = p->Compute(p->Par, &VStock);

    if (k == OP_Exercise_Dates - 1)
    {
        AuxScal_internal = 0.;
    }
    else
    {
        if (AL_PayOff_As_Regressor <= k)
        {
            reg = MGET(BasisL2, k, 0);
            AuxScal_internal = AuxOption_internal * reg;

            for (l = 1; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal_internal += pnl_basis_i(Basis, pat
            }
        }
        else
        {
            AuxScal_internal = 0.;
        }
    }
}

```

```

        for (l = 0; l < AL_Basis_Dimension; l++)
        {
            reg = MGET(BasisL2, k, l);
            AuxScal_internal += pnl_basis_i(Basis, pat
        }
    }
    estimate_mean[i] += MAX(AuxScal_internal, AuxOption_in
}
}
}

//Put in Paths the paths kept in memory
for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
{
    for (l = 0; l < BS_Dimension; l++)
    {
        Paths[i * BS_Dimension + l] = MGET(S, k, i * BS_Dimension + l)
        if (use_normalised_regressor)
        {
            PathsN[i * BS_Dimension + l] = MGET(SN, 1, i * BS_Dimensio
            paths = PathsN;
        }
    }
}

n0 = 0;
for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
{
    if ((GET(ZERO, i) == 1) || ((GET(ZERO, i) == 0) && (n0 < lz)))
    {
        if ((GET(ZERO, i) == 0) && (n0 < lz))    n0++;

        VStock.size = BS_Dimension;
        VStock.array = &(Paths[i * BS_Dimension]);
        AuxOption = p->Compute(p->Par, &VStock);

        if (k == OP_Exercise_Dates - 1)

```



```

    {
        AuxScal = 0.;
    }
else
    {
        if (AL_PayOff_As_Regressor <= k)
        {
            reg = MGET(BasisL2, k, 0);
            AuxScal = AuxOption * reg;
            for (l = 1; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimen
            }
        }
        else
        {
            AuxScal = 0.;
            for (l = 0; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimen
            }
        }
    }

if (Auxopt[i] <= Auxscal[i])
{
    PI[i] += MAX(AuxOption, AuxScal) * DiscountFactor[k] - L[i]
}

else
{
    PI[i] += MAX(AuxOption, AuxScal) * DiscountFactor[k] - est
}

L[i] = MAX(AuxOption, AuxScal);
Auxopt[i] = AuxOption;
Auxscal[i] = AuxScal;
MM[i] = MAX(MM[i], AuxOption * DiscountFactor[k] - PI[i]);
}

```

```

    }

}

for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
{
    if ((GET(ZERO, i) == 0) && (n0 < 1z))
    {
        n0++;
        E0 += MM[i];
    }
    if (GET(ZERO, i) == 1)
    {
        n1++;
        E1 += MM[i];
    }
}
E0 *= (double)(AL_MonteCarlo_Iterations_Dual - n1) / (double)n0;

*Dual_Price = E0 + E1;
*Dual_Price /= (double)AL_MonteCarlo_Iterations_Dual;
*Dual_Price += AL_FPrice;
}

//No boundary distance grouping procedure
else
{

    // Martingale initialization according to Andersen/Broadie Algorithm

    //Initial Step : k=0

    for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
    {
        AuxOption = p->Compute(p->Par, BS_Spot);
        AuxScal = MGET(BasisL2, 0, 0);
        L[i] = AL_FPrice;
        Auxopt[i] = AuxOption;
        Auxscal[i] = AuxScal;
        PI[i] = AL_FPrice;
    }
}

```

```

    MM[i] = AuxOption * DiscountFactor[0] - PI[i];
}

//Building the martingale thanks to Andersen/Broadie algorithm, from step

for (k = 1; k <= OP_Exercice_Dates - 1; k++)
{
    for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
    {
        for (l = 0; l < BS_Dimension; l++)
        {
            LET(Spot_Internal, l) = Paths[i * BS_Dimension + l];
        }

        if (Auxopt[i] > Auxscal[i] &&
            Auxopt[i] > European_Lower_bound(Spot_Internal, p, OP_Maturity,
                                              BS_Dividend_Rate, BS_Volatili
        {
            estimate_mean[i] = 0.;

            if (AL_Antithetic)
                /*here, the brownian path is computed with antithetic paths*
                Compute_Brownian_Paths_A(Brownian_PathsInternal, sqrt((double)
                                          BS_Dimension, AL_MonteCarlo_Iterati
                                          generator);
            else
                Compute_Brownian_Paths(Brownian_PathsInternal, sqrt((double)
                              BS_Dimension, AL_MonteCarlo_Iteration
                              generator);

            /*computation of the BlackScholes paths at time k related to B

            BS_Forward_Path(PathsInternal, Brownian_PathsInternal, Spot_In
                           (double) Step, AL_MonteCarlo_Iterations_Dual_I

            if (use_normalised_regressor)
            {

```

```

Compute_Inv_Sqrt_BS_Dispersion((double)Step, BS_Dimension,
                                BS_Interest_Rate, BS_Divide

NormalisedPaths(PathsInternal, PathsNInternal, AL_MonteCar
pathsInternal = PathsNInternal;
}

for (m = 0; m < AL_MonteCarlo_Iterations_Dual_Internal; m++)
{
    VStock.size = BS_Dimension;
    VStock.array = &(PathsInternal[m * BS_Dimension]);
    AuxOption_internal = p->Compute(p->Par, &VStock);

    if (k == OP_Exercise_Dates - 1)
    {
        AuxScal_internal = 0.;
    }
    else
    {
        if (AL_PayOff_As_Regressor <= k)
        {
            reg = MGET(BasisL2, k, 0);
            AuxScal_internal = AuxOption_internal * reg;

            for (l = 1; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal_internal += pnl_basis_i(Basis, pathsIn
            }
        }
        else
        {
            AuxScal_internal = 0.;
            for (l = 0; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal_internal += pnl_basis_i(Basis, pathsIn
            }
        }
    }
}

```

```

        }
        estimate_mean[i] += MAX(AuxScal_internal, AuxOption_intern
    }
}

if (AL_Antithetic)
    /*here, the brownian path is computed with antithetic paths*/
    Compute_Brownian_Paths_A(Brownian_Paths, sqrt((double)k * Step),
        BS_Dimension, AL_MonteCarlo_Iterations_Dual,
        generator);
else
    Compute_Brownian_Paths(Brownian_Paths, sqrt((double)k * Step),
        BS_Dimension, AL_MonteCarlo_Iterations_Dual,
        generator);

/*computation of the BlackScholes paths at time k related to Brownian
BS_Forward_Path(Paths, Brownian_Paths, BS_Spot->array, k * Step, AL_Mo

if (use_normalised_regressor)
{
    Compute_Inv_Sqrt_BS_Dispersion(k * Step, BS_Dimension, BS_Spot,
        BS_Interest_Rate, BS_Dividend_Rate)

    NormalisedPaths(Paths, PathsN, AL_MonteCarlo_Iterations_Dual, BS_D
    paths = PathsN;
}

for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
{
    VStock.size = BS_Dimension;
    VStock.array = &(Paths[i * BS_Dimension]);
    AuxOption = p->Compute(p->Par, &VStock);

    if (k == OP_Exercice_Dates - 1)
    {
        AuxScal = 0.;
    }
    else

```

```

    {
        if (AL_PayOff_As_Regressor <= k)
        {
            reg = MGET(BasisL2, k, 0);
            AuxScal = AuxOption * reg;
            for (l = 1; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimension, l);
            }
        }
        else
        {
            AuxScal = 0.;
            for (l = 0; l < AL_Basis_Dimension; l++)
            {
                reg = MGET(BasisL2, k, l);
                AuxScal += pnl_basis_i(Basis, paths + i * BS_Dimension, l);
            }
        }
    }

    if (Auxopt[i] <= Auxscal[i])
    {
        PI[i] += MAX(AuxOption, AuxScal) * DiscountFactor[k] - L[i] *
    }

    else
    {
        PI[i] += MAX(AuxOption, AuxScal) * DiscountFactor[k] - estimat
    }

    L[i] = MAX(AuxOption, AuxScal);
    Auxopt[i] = AuxOption;
    Auxscal[i] = AuxScal;
    MM[i] = MAX(MM[i], AuxOption * DiscountFactor[k] - PI[i]);
}

for (i = 0; i < AL_MonteCarlo_Iterations_Dual; i++)
{

```

```

        *Dual_Price += MM[i] / (double)AL_MonteCarlo_Iterations_Dual;
    }

    *Dual_Price += AL_FPrice;

}

/*memory liberation*/
AnBrB_Liberation();
End_BS();
pnl_basis_free(&Basis);

return OK;
}

int CALC(MC_AndersenBroadieND)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r;
    double *BS_cor;
    int i, res;
    PnlVect *divid = pnl_vect_create(ptMod->Size.Val.V_PINT);
    PnlVect *spot, *sig;

    pipo = fopen("poo.txt", "w");
    spot = pnl_vect_compact_to_pnl_vect(ptMod->S0.Val.V_PNLVECTCOMPACT);
    sig = pnl_vect_compact_to_pnl_vect(ptMod->Sigma.Val.V_PNLVECTCOMPACT);
    for (i = 0; i < ptMod->Size.Val.V_PINT; i++)
        pnl_vect_set(divid, i,
                     log(1. + pnl_vect_compact_get(ptMod->Divid.Val.V_PNLVECTCOMPACT, i)));

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

    if ((BS_cor = malloc(ptMod->Size.Val.V_PINT * ptMod->Size.Val.V_PINT * sizeof(
        return MEMORY_ALLOCATION_FAILURE;
    for (i = 0; i < ptMod->Size.Val.V_PINT * ptMod->Size.Val.V_PINT; i++)
        BS_cor[i] = ptMod->Rho.Val.V_DOUBLE;
    for (i = 0; i < ptMod->Size.Val.V_PINT; i++)

```

```

    BS_cor[i * ptMod->Size.Val.V_PINT + i] = 1.0;

    res = AnBrB(spot,
                ptOpt->PayOff.Val.V_NUMFUNC_ND,
                ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                r, divid, sig,
                BS_cor,
                Met->Par[0].Val.V_LONG,
                Met->Par[1].Val.V_ENUM.value,
                Met->Par[2].Val.V_ENUM.value,
                Met->Par[3].Val.V_INT,
                Met->Par[4].Val.V_INT,
                Met->Par[5].Val.V_ENUM.value,
                Met->Par[6].Val.V_ENUM.value,
                Met->Par[7].Val.V_ENUM.value,
                Met->Par[8].Val.V_LONG,
                Met->Par[9].Val.V_LONG,
                Met->Par[10].Val.V_ENUM.value,
                &(Met->Res[0].Val.V_DOUBLE));
    pnl_vect_free(&divid);
    free(BS_cor);
    pnl_vect_free(&spot);
    pnl_vect_free(&sig);
    fclose(pipo);
    return res;
}

static int CHK_OPT(MC_AndersenBroadieND)(void *Opt, void *Mod)
{
    Option *ptOpt = (Option *)Opt;
    TYPEOPT *opt = (TYPEOPT *) (ptOpt->TypeOpt);
    if ((opt->EuOrAm).Val.V_BOOL == AMER)
        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_LONG = 50000;
    Met->Par[1].Val.V_ENUM.value = 0;
    Met->Par[1].Val.V_ENUM.members = &PremiaEnumMCRNGs;
    Met->Par[2].Val.V_ENUM.value = 0;
    Met->Par[2].Val.V_ENUM.members = &PremiaEnumBasis;
    Met->Par[3].Val.V_INT = 3;
    Met->Par[4].Val.V_INT = 10;
    Met->Par[5].Val.V_ENUM.value = 1;
    Met->Par[5].Val.V_ENUM.members = &PremiaEnumBool;
    Met->Par[6].Val.V_ENUM.value = 0;
    Met->Par[6].Val.V_ENUM.members = &PremiaEnumBool;
    Met->Par[7].Val.V_ENUM.value = 0;
    Met->Par[7].Val.V_ENUM.members = &PremiaEnumBool;
    Met->Par[8].Val.V_LONG = 500;
    Met->Par[9].Val.V_LONG = 100;
    Met->Par[10].Val.V_ENUM.value = 0;
    Met->Par[10].Val.V_ENUM.members = &PremiaEnumBool;
}
return OK;
}

PricingMethod MET(MC_AndersenBroadieND) =
{
    "MC_AndersenBroadie_ND",
    { {"N iterations Primal", LONG, {100}, ALLOW},
      {"RandomGenerator", ENUM, {0}, ALLOW},
      {"Basis", ENUM, {1}, ALLOW},
      {"Approximation order", INT, {100}, ALLOW},
      {"Number of Exercise Dates", INT, {100}, ALLOW},
      {"Use Payoff as Regressor", ENUM, {1}, ALLOW},
      {"Use Antithetic Variables", ENUM, {1}, ALLOW},
      {"Use Normalised Regressors", ENUM, {0}, ALLOW},
      {"N iterations Dual", LONG, {100}, ALLOW},
      {"N iterations Dual Internal", LONG, {100}, ALLOW},
      {"Use Boundary Distance Grouping as Regressor", ENUM, {1}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_AndersenBroadieND),
    { {"Price", DOUBLE, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    }
}

```

```
    },  
    CHK_OPT(MC_AndersenBroadieND),  
    CHK_mc,  
    MET(Init)  
};
```