

## [Help](#)

```
#include <stdlib.h>
#include <stdio.h>
#include <
href../../../../common/math/cdo/cdo_math_h_src.pdfmath.h>

#include "
href../../../../mod/bsnd/bsnd_stdnd/bsnd_stdnd_h_src.pdfbsnd_stdnd.h"
#include "
href../../../../common/enums_h_src.pdfenums.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_basis.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
static int CHK_OPT(MC_BSDE_Labart)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_BSDE_Labart)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

static double (*driverf)(double y, double r, double penal, double payoff);
static double *convert_tx(double t, PnlVect *x);
static double approx_fonction(PnlBasis *B, PnlVect *alpha, double *tx);

/**
 * Driver for European options
 *
 * @param y value of Y in the BSDE
 * @param r instantaneous interest rates
 * @param penal unused but necessary to ensure driver_euro and driver_amer
 * take the same args.
 * @param payoff unused but necessary to ensure driver_euro and driver_amer
 * take the same args.
 *
 */
```

```

    * @return
    */
static double driverf_euro(double y, double r, double penal, double payoff)
{
    return -r * y;
}

/**
 * Driver for American options
 *
 * @param y value of Y in the BSDE
 * @param r instantaneous interest rates
 * @param penal weight of the penalisation
 * @param payoff payoff at current values of (time, space)
 *
 * @return
 */
static double driverf_amer(double y, double r, double penal, double payoff)
{
    return -r * y - penal * MIN(y - payoff, 0.0);
}

/*
 * condition terminale g
 */
static double payoffg(PnlVect *x, NumFunc_nd *p)
{
    return p->Compute(p->Par, x);
}

/*
 * Simulation of Black Scholes
 */

/**
 * Construit la factorisée de Cholesky (triangulaire inférieure) de la
 * matrice de corrélation (avec des corr partout et une diagonale de 1)
 *
 * @param correl (out) factorisée de Cholesky de la matrice de corrélation
 * @param d dimension du modèle
 * @param corr corrélation entre les copposantes

```

```

    */
static void init_correl(PnlMat **correl, int d, double corr)
{
    *correl = pnl_mat_create_from_double(d, d, corr);
    pnl_mat_set_diag(*correl, 1, 0.);
    pnl_mat_chol((*correl));
}

/*
 * fonction b : attention b est donné sous la proba historique
 */
static void b(PnlVect *res, double t, const PnlVect *x, double r, const PnlVect
{
    pnl_vect_clone(res, divid);
    pnl_vect_minus(res);
    pnl_vect_plus_double(res, r);
    pnl_vect_mult_vect_term(res, x); //res=mu x
}

/*
 * vecteur  $\sigma^i * X^i$ 
 */
static void vol(PnlVect *res, double t, const PnlVect *x, PnlVect *sigma)
{
    pnl_vect_clone(res, x);
    pnl_vect_mult_vect_term(res, sigma);
}

/**
 * Calcule la matrice  $(\sigma^i * X^i * \text{Ligne}(\text{correl}, i))_i$ 
 *
 * @param res (out) contient la matrice calculée
 * @param t l'instant auquel on fait le calcul
 * @param x valeur du sous-jacent à l'instant t
 * @param sigma le vecteur de volatilité
 * @param correl la factorise de Cholesky de la matrice de corrélation
 */
static void vol_correl(PnlMat *res, double t, const PnlVect *x, PnlVect *sigma,
{
    int i, j;
    PnlVect *volatility;

```

```

int d = x->size;
volatility = pnl_vect_create(0);
vol(volatility, t, x, sigma);
pnl_mat_resize(res, d, d);
pnl_mat_set_double(res, 0.);

for (i = 0; i < d; i++)
{
    double sigma_i = pnl_vect_get(volatility, i);
    for (j = 0; j < i + 1; j++)
    {
        pnl_mat_set(res, i, j, sigma_i * pnl_mat_get(correl, i, j));
    }
}
pnl_vect_free(&volatility);
}

/**
 * Cette fonction simule une valeur de S_t connaissant S_t0 dans un mdoèle
 * de Black-Scholes
 *
 * @param S (out) contient une simulation du sous-jacent à l'instant t
 * sachant la valeur x0 à l'instant t0
 * @param x0 valeur du sous-jacent à l'instant t0
 * @param t0 instant initial
 * @param t instant auquel on souhaite simuler S
 * @param r taux d'intérêt
 * @param sigma vecteur de volatilités
 * @param correl matrice de corrélation
 * @param divid vecteur de dividende
 * @param rng générateur
 */
static void Simul_Asset(PnlVect *S, const PnlVect *x0, double t0, double t,
                        double r, PnlVect *sigma, PnlMat *correl, PnlVect *divid,
                        PnlRng *rng)
{
    static PnlVect *G = NULL;
    int i, d;
    double dt, sqrt_dt;
    if (G == NULL)
    {

```

```

        G = pnl_vect_create(0);
    }
    /* call with S=NULL to free all static variables */
    if (S == NULL)
    {
        pnl_vect_free(&G);
        return;
    }
    d = x0->size;
    pnl_vect_clone(S, x0);

    pnl_vect_rng_normal(G, d, rng);
    dt = t - t0;
    sqrt_dt = sqrt(t - t0);
    for (i = 0 ; i < d ; i++)
    {
        double sig = GET(sigma, i);
        double div_i = GET(divid, i);
        PnlVect Li = pnl_vect_wrap_mat_row(correl, i);
        LET(S, i) = GET(x0, i) * exp(((r - div_i) - sig * sig / 2) * dt
                                     + sig * sqrt_dt * pnl_vect_scalar_prod(&Li, G
                                     )
        }
    }

/**
 * Calcule les bornes du support
 *
 * @param V_inf vecteur des bornes inférieures
 * @param V_sup vecteur des bornes supérieures
 * @param y vecteur des spots
 * @param r taux d'intérêt
 * @param sigma vecteur des volatilités
 * @param mu vecteur des tendances
 * @param T maturité
 */
static void support(PnlVect *V_inf, PnlVect *V_sup, const PnlVect *y, double r,
                   const PnlVect *sigma, const PnlVect *divid, double T)
{
    PnlVect *log_y;
    double min_sigma, max_sigma, min_divid, max_divid;
    pnl_vect_minmax(&min_sigma, &max_sigma, sigma);

```

```

    pnl_vect_minmax(&min_divid, &max_divid, divid);
    log_y = pnl_vect_create(y->size);
    pnl_vect_map(V_inf, y, log);
    pnl_vect_clone(V_sup, V_inf);
    pnl_vect_plus_double(V_inf, (r - max_divid - max_sigma * max_sigma * 0.5)*T -
    pnl_vect_plus_double(V_sup, (r - min_divid - min_sigma * min_sigma * 0.5)*T +
    pnl_vect_map_inplace(V_inf, exp);
    pnl_vect_map_inplace(V_sup, exp);
    pnl_vect_free(&log_y);
}

/*
 * W operator
 */

/*
 * fonction W_chapeau initiale de l'algorithme, lorsque u0=0. M nb de tirages MC
 * nb de points de discrétisation du SE // X_new de taille d
 */
static double W_chapeau_init(double T_alea_new, PnlVect *X_new, int M, double
                                T, double r, PnlVect *sigma, PnlMat *correl,
                                PnlVect *divid, PnlRng *rng, NumFunc_nd *p, double
{
    double sum, Um, payoff_t;
    PnlVect *St, *ST, *Z;
    int m;
    sum = 0.0;
    St = pnl_vect_create(0);
    ST = pnl_vect_create(0);
    Z = pnl_vect_create_from_zero(X_new->size);
    for (m = 0; m < M; m++)
    {
        Um = pnl_rng_uni_ab(T_alea_new, T, rng);
        Simul_Asset(St, X_new, T_alea_new, Um, r, sigma, correl, divid, rng);
        Simul_Asset(ST, St, Um, T, r, sigma, correl, divid, rng);
        payoff_t = payofff(St, p);
        sum += (T - T_alea_new) * driverf(0.0, r, penal, payoff_t) + payofff(ST, p
    }
}

```

```

    pnl_vect_free(&St);
    pnl_vect_free(&ST);
    pnl_vect_free(&Z);
    return sum / M;
}

/**
 * Calule la correction c par une boucle Monte-Carlo
 *
 * @param M nombre de tirages MC
 * @param N nombre de pas du schéma d'Euler
 * @param T maturité de l'option
 * @param X_old anciens points aléatoires en espace
 * @param T_alea_old anciens points aléatoires en temps
 * @param X_new nouveaux points aléatoires en espace
 * @param T_alea_new nouveaux points aléatoires en temps
 * @param r taux d'interet
 * @param divid taux de dividende
 * @param K strike
 * @param sigma vecteur de volatilités
 * @param correl matrice de correlation
 * @param rng generateur à utiliser
 */
static double W_chapeau(double T_alea_new, PnlVect *X_new, int M,
                        double T, double r, PnlVect *sigma, PnlMat *correl,
                        PnlVect *divid, PnlBasis *basis, PnlVect *alpha,
                        PnlRng *rng, NumFunc_nd *p, double penal)
{
    double sum, sum1, D_0, D_t, Um, *tx, payoff_t;
    PnlVect *St, *ST, *D_x, *grad, *drift;
    PnlMat *D_xx, *volatility, *tmp_vol, *tmp_vol_square;
    int m;
    int d = X_new->size;
    sum = 0.0;
    St = pnl_vect_create(0);
    ST = pnl_vect_create(0);
    D_x = pnl_vect_create(0);
    D_xx = pnl_mat_create(0, 0);
    grad = pnl_vect_create(0);
    drift = pnl_vect_create(0);

```

```

tmp_vol = pnl_mat_create(0, 0);
tmp_vol_square = pnl_mat_create(0, 0);
volatility = pnl_mat_create(0, 0);
for (m = 0; m < M; m++)
{
    sum1 = 0.0;
    Um = pnl_rng_uni_ab(T_alea_new, T, rng);
    Simul_Asset(St, X_new, T_alea_new, Um, r, sigma, correl, divid, rng);
    Simul_Asset(ST, St, Um, T, r, sigma, correl, divid, rng);
    tx = convert_tx(Um, St);

    pnl_basis_eval_derivs(basis, alpha, tx, &D_0, grad, tmp_vol);
    D_t = PNL_GET(grad, 0);
    pnl_vect_extract_subvect(D_x, grad, 1, d);
    pnl_mat_extract_subblock(D_xx, tmp_vol, 1, d, 1, d);

    vol_correl(volatility, Um, St, sigma, correl);
    b(drift, Um, St, r, divid);
    pnl_mat_clone(tmp_vol, volatility);
    pnl_mat_sq_transpose(volatility);
    pnl_mat_mult_mat_inplace(tmp_vol_square, tmp_vol, volatility);
    pnl_mat_mult_mat_term(tmp_vol_square, D_xx);
    payoff_t = payofff(St, p);
    sum1 = driverf(D_0, r, penal, payoff_t) + D_t + pnl_vect_scalar_prod(drift,
        + 0.5 * pnl_mat_sum(tmp_vol_square);
    tx = convert_tx(T, ST);
    sum1 = (T - T_alea_new) * sum1 + payofff(ST, p) - approx_fonction(basis, a
    sum += sum1;
}
pnl_vect_free(&St);
pnl_vect_free(&ST);
pnl_vect_free(&D_x);
pnl_mat_free(&D_xx);
pnl_vect_free(&grad);
pnl_vect_free(&drift);
pnl_mat_free(&tmp_vol);
pnl_mat_free(&tmp_vol_square);
pnl_mat_free(&volatility);
return sum / M;
}

```



```

/**
 * Calule la correction c par une boucle Monte-Carlo
 *
 * @param M nombre de tirages MC
 * @param N nombre de pas du schéma d'Euler
 * @param T maturité de l'option
 * @param X_old anciens points aléatoires en espace
 * @param T_alea_old anciens points aléatoires en temps
 * @param X_new nouveaux points aléatoires en espace
 * @param T_alea_new nouveaux points aléatoires en temps
 * @param r taux d'interet
 * @param divid taux de dividende
 * @param K strike
 * @param sigma vecteur de volatilités
 * @param correl matrice de correlation
 * @param rng generateur à utiliser
 */
static double W_chapeau_wrapper(PnlVect *TX_new, int M, double T, double r, PnlVect
                                *sigma, PnlMat *correl, PnlVect *divid,
                                PnlBasis *basis, PnlVect *alpha, PnlRng *rng,
                                NumFunc_nd *p, double penal)
{
    double T_new = PNL_GET(TX_new, 0);
    PnlVect X_new = pnl_vect_wrap_subvect(TX_new, 1, TX_new->size - 1);
    if (alpha->size == 0)
    {
        return W_chapeau_init(T_new, &X_new, M, T, r, sigma, correl, divid, rng, p);
    }
    else
    {
        return W_chapeau(T_new, &X_new, M, T, r, sigma, correl, divid, basis, alpha, rng, p, penal);
    }
}

/**
 * Approximation stuff
 */

```

```

static double *tx = NULL;

static void init_pol_approx(int d)
{
    if (tx != NULL)
    {
        perror("Polynomial approximation already initialized\ n");
        abort();
    }
    tx = malloc(sizeof(double) * (d + 1));
}

static void free_pol_approx()
{
    if (tx == NULL) return;
    free(tx);
    tx = NULL;
}

static double *convert_tx(double t, PnlVect *x)
{
    tx[0] = t;
    memcpy(tx + 1, x->array, x->size * sizeof(double));
    return tx;
}

static double approx_fonction(PnlBasis *B, PnlVect *alpha, double *tx)
{
    double res;
    res = pnl_basis_eval(B, alpha, tx);
    return res;
}

static void derive_x_approx_fonction(PnlVect *res, PnlBasis *B, PnlVect *alpha,
{
    int i;
    pnl_vect_resize(res, d);
    for (i = 0; i < d; i++)
        pnl_vect_set(res, i, pnl_basis_eval_D(B, alpha, tx, i + 1));
}

```

```

/**
 * Calule la solution et sa dérivée en espace (le prix et le delta) au point
 * (T_disc, X_disc)
 *
 * @param res prix au point (T_disc, X_disc)
 * @param res_x delta au point (T_disc, X_disc)
 * @param M nombre de tirages MC
 * @param Np nombre de points de la grille
 * @param T_disc valeur de t auquel on calcule la solution
 * @param X_disc valeur de x à laquelle on calcule la solution
 * @param m total degree of the polynomial approximation
 * @param Kit nombre d'itérations globales de l'algorithme
 * @param T maturité de l'option
 * @param r taux d'interet
 * @param divid taux de dividende
 * @param K strike
 * @param sigma volatilités
 * @param correl matrice de correlation
 * @param V_inf borne inférieur du domaine spatiale
 * @param V_sup borne supérieure du domaine spatiale
 * @param basis base de polynomes
 * @param rng gnerateur à utiliser
 */
static void algorithme(double *res, PnlVect *res_x, int M, int Np, double
                    T_disc, PnlVect *X_disc, int Kit, double
                    T, double r, PnlVect *divid, PnlVect *sigma,
                    PnlMat *correl, PnlVect *V_inf, PnlVect *V_sup,
                    PnlBasis *basis, PnlRng *rng, NumFunc_nd *p, double penal
{
    int i, j, k, d;
    double U_dec_i, W_i, sum_W_U_i, *tx;
    PnlVect *T_alea_old, *sum_W_U, *U, *U_dec, *W, *alpha_u;
    PnlMat *X_old, *TX;

    d = X_disc->size;

    pnl_vect_resize(res_x, d);

```

```

sum_W_U = pnl_vect_create(Np);
W = pnl_vect_create(Np);
U = pnl_vect_create(Np);
X_old = pnl_mat_create(Np, d);
TX = pnl_mat_create(Np, d + 1);
U_dec = pnl_vect_create(Np);
T_alea_old = pnl_vect_create(Np);
alpha_u = pnl_vect_new();

//T_alea_new=pnl_vect_create(Np);
/* simulation des (t_i, x_i) */
pnl_vect_rng_uni(T_alea_old, Np, 0.0, T, rng);
pnl_mat_rng_uni(X_old, Np, d, V_inf, V_sup, rng);

//on place en composante 0 de T_alea_old T_disc
//on place sur la 1ere ligne de X_old X_disc
pnl_vect_set(T_alea_old, 0, T_disc);
for (i = 0; i < d; i++) pnl_mat_set(X_old, 0, i, pnl_vect_get(X_disc, i));

for (i = 0; i < Np; i++)
{
    MLET(TX, i, 0) = GET(T_alea_old, i);
    for (j = 0; j < d; j++)
    {
        MLET(TX, i, j + 1) = MGET(X_old, i, j);
    }
}
// DEBUT DES ITERATIONS

/* alpha_u must be of size 0 to detect it is iteration 0 */
for (k = 0; k < Kit; k++)
{
    /* boucle à paralléliser */
    for (i = 0 ; i < Np ; i++)
    {
        PnlVect TX_i = pnl_vect_wrap_mat_row(TX, i);
        if (alpha_u->size == 0)
        {
            U_dec_i = 0;

```

```

        }
    else
    {
        U_dec_i = pnl_basis_eval(basis, alpha_u, TX_i.array);
    }
    W_i = W_chapeau_wrapper(&TX_i, M, T, r, sigma, correl, divid,
                           basis, alpha_u, rng, p, penal);
    sum_W_U_i = U_dec_i + W_i;
    pnl_vect_set(U_dec, i, U_dec_i);
    pnl_vect_set(W, i, W_i);
    pnl_vect_set(sum_W_U, i, sum_W_U_i);
}
pnl_basis_fit_ls(basis, alpha_u, TX, sum_W_U);
}

*res = pnl_vect_get(sum_W_U, 0);
tx = convert_tx(T_disc, X_disc);
derive_x_approx_fonction(res_x, basis, alpha_u, tx, d);
free_pol_approx();
pnl_vect_free(&T_alea_old);
pnl_mat_free(&X_old);
pnl_mat_free(&TX);
pnl_vect_free(&sum_W_U);
pnl_vect_free(&U);
pnl_vect_free(&U_dec);
pnl_vect_free(&W);
pnl_vect_free(&alpha_u);
}

/*
 * fonction principale
 */
static void mc_bsde(PnlVect *spot, double T, double r, PnlVect *divid, PnlVect
                   *sigma, double corr, int M, int Np, int m, int basis_type,
                   int K_it, PnlRng *rng, NumFunc_nd *p, double penal,
                   double *prix, PnlVect *delta)
{
    int i;
    PnlVect *V_inf, *V_sup, *Dmin, *Dmax;
    PnlMat *correl;
    int d = spot->size;
    PnlBasis *basis;

```

```

/* init */
V_inf = pnl_vect_create(d);
V_sup = pnl_vect_create(d);
init_correl(&correl, d, corr);
support(V_inf, V_sup, spot, r, sigma, divid, T);

init_pol_approx(d);
basis = pnl_basis_create_from_degree(basis_type, m, d + 1);
Dmin = pnl_vect_create(d + 1);
Dmax = pnl_vect_create(d + 1);
PNL_LET(Dmin, 0) = 0.;
PNL_LET(Dmax, 0) = T;
for (i = 0 ; i < d ; i++)
{
    PNL_LET(Dmin, i + 1) = PNL_GET(V_inf, i);
    PNL_LET(Dmax, i + 1) = PNL_GET(V_sup, i);
}
pnl_basis_set_domain(basis, Dmin, Dmax);
pnl_vect_free(&Dmin);
pnl_vect_free(&Dmax);

algorithm(prix, delta, M, Np, 0, spot, K_it, T, r, divid, sigma,
          correl, V_inf, V_sup, basis, rng, p, penal);

/* free some static variables */
Simul_Asset(NULL, NULL, 0, 0, 0, NULL, NULL, NULL, NULL);
pnl_vect_free(&V_inf);
pnl_vect_free(&V_sup);
pnl_mat_free(&correl);
pnl_basis_free(&basis);
}

int CALC(MC_BSDE_Labart)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r;
    int i, degree;
    PnlRng *rng;
    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;
rng = PnlRngArray[Met->Par[0].Val.V_ENUM.value];
pnl_rng_sseed(rng, 0);
degree = Met->Par[2].Val.V_INT;

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_bsde(spot,
    ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
    r, divid, sig,
    ptMod->Rho.Val.V_DOUBLE,
    MIN(Met->Par[4].Val.V_INT, 30000), /* M */
    MIN(Met->Par[3].Val.V_INT, 2000), /* Np */
    degree,
    Met->Par[1].Val.V_ENUM.value,
    5, /* K_it */
    rng,
    ptOpt->PayOff.Val.V_NUMFUNC_ND,
    Met->Par[5].Val.V_DOUBLE,
    &(Met->Res[0].Val.V_DOUBLE), Met->Res[1].Val.V_PNLVECT);
pnl_vect_free(&divid);
return OK;
}

static int CHK_OPT(MC_BSDE_Labart)(void *Opt, void *Mod)
{
    Option *ptOpt = (Option *)Opt;

    if ((strcmp(ptOpt->Name, "CallBasketEuro_nd") == 0) ||
        (strcmp(ptOpt->Name, "PutBasketEuro_nd") == 0) ||
        (strcmp(ptOpt->Name, "PutBasketAmer_nd") == 0))
        return OK;

    return WRONG;
}

```

```

}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    TYPEOPT *opt = (TYPEOPT *) (Opt->TypeOpt);
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->Res[1].Val.V_PNLVECT = NULL;
        Met->Par[0].Val.V_ENUM.value = 0;
        Met->Par[0].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[1].Val.V_ENUM.value = 0;
        Met->Par[1].Val.V_ENUM.members = &PremiaEnumBasis;
        Met->Par[2].Val.V_INT = 3;
        Met->Par[3].Val.V_INT = 500;
        Met->Par[4].Val.V_INT = 100;
        if ((opt->EuOrAm).Val.V_BOOL == EURO)
        {
            driverf = driverf_euro;
            Met->Par[5].Val.V_DOUBLE = 0.;
        }
        else
        {
            driverf = driverf_amer;
            Met->Par[5].Val.V_DOUBLE = 1.5;
        }
    }
    /* some initialisation */
    if (Met->Res[1].Val.V_PNLVECT == NULL)
        Met->Res[1].Val.V_PNLVECT = pnl_vect_create(opt->Size.Val.V_PINT);
    else
        pnl_vect_resize(Met->Res[1].Val.V_PNLVECT, opt->Size.Val.V_PINT);

    return OK;
}

PricingMethod MET(MC_BSDE_Labart) =
{
    "MC_BSDE_Labart_nd",

```



```

{
  {"RandomGenerator", ENUM, {0}, ALLOW},
  {"Basis", ENUM, {0}, ALLOW},
  {"Degree", INT, {0}, ALLOW},
  {"Nb of Points", INT, {0}, ALLOW},
  {"nb of MC", INT, {0}, ALLOW},
  {"Penal", DOUBLE, {0}, ALLOW, SETABLE},
  {" ", PREMIA_NULLTYPE, {0}, FORBID}
},
CALC(MC_BSDE_Labart),
{ {"Price", DOUBLE, {100}, FORBID}, {"Delta", PNLVECT, {1}, FORBID},
  {" ", PREMIA_NULLTYPE, {0}, FORBID}
},
CHK_OPT(MC_BSDE_Labart),
CHK_ok,
MET(Init)
};

```