

## Help

```
#include "bs1d_default_std.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_basis.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_list.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2014+2) //The "#els

static int CHK_OPT(MC_Branch_HLabordere)(void *Opt, void *Mod)
{
    return NONACTIVE;
}

int CALC(MC_Branch_HLabordere)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

/**
 * Characteristics of a product
 */
typedef struct _Product Product;
struct _Product
{
    double r; /*!< interest rate */
    double S0; /*!< spot */
    double K; /*!< strike */
    double T; /*!< maturity */
    double beta; /*!< Poisson intensity CDS spread */
    double sigma; /*!< volatility */
    double Recovery; /*!< Recovery rate*/
};

//vecteur des probabilités p_k optimales issu de (18)
static void proba(PnlVect *p, const PnlVect *a)
```

```

{
    int i;
    double s;
    s = pnl_vect_norm_one(a);
    for (i = 0 ; i < p->size ; i++)
        LET(p, i) = fabs(GET(a, i)) / s;
}

//calcul des coefficients a_k/|a|
static void coeff_PDE2(PnlVect *a_p, const PnlVect *a, const PnlVect *p)
{
    int i;
    double a_1 = pnl_vect_norm_one(a);
    for (i = 0; i < a->size; i++)
    {
        LET(a_p, i) = a_1 * PNL_SIGN(GET(a, i));
    }
}

//calcul des coefficients a_k/p_k
static void coeff_PDE1(PnlVect *a_p, const PnlVect *a, const PnlVect *p)
{
    int i;
    for (i = 0; i < a->size; i++)
    {
        if (GET(a, i) == 0) LET(a_p, i) = 1.0;
        else LET(a_p, i) = GET(a, i) / GET(p, i);
    }
}

//initialisation du vecteur des positions
static void init(PnlVect *Z, double t, PnlRng *rng, const Product *P)
{
    double g = pnl_rng_normal(rng);
    LET(Z, 0) = P->S0 * exp((P->r - 0.5 * pow(P->sigma, 2)) * t + P->sigma * sqrt(
}

//payoff
static double psi(double z, const Product *P)
{
    if (z > P->K) return -1.0;
}

```

```
else return 1.0;

//return MAX(0,z-P->K);;
}

//tirage d'une variable aléatoire entière à valeur dans [0,M] de loi de proba p
static int tirage_proba_p(const PnlVect *p, PnlRng *rng)
{
    PnlVect *pcopy;
    int j = 0;
    double u;
    pcopy = pnl_vect_copy(p);
    pnl_vect_cumsum(pcopy);
    u = pnl_rng_uni(rng);
    while ((j < pcopy->size) && (GET(pcopy, j) < u))
    {
        j++;
    }
    pnl_vect_free(&pcopy);
    return j;
}

//tirage d'une loi uniforme sur [0,n-1]
static int tirage_uni(int n, PnlRng *rng)
{
    double u = pnl_rng_uni(rng);
    return floor(n * u);
}

//nb de particules à l'étage k :  $\sum_{k=0}^M (\omega(k) * (k-1)) + 1$ 
static int somme(PnlVect *omega)
{
    int i;
    int s = 0;
    for (i = 0; i < omega->size; i++)
    {
        s = s + GET(omega, i) * (i - 1);
    }
    return s + 1;
}
```

```

//actualisation de
// omega : omega(k) = nb de branches de l'arbre ayant donné k fils
//Z : Z(i) = position de la particule i
//n_fils, quel_fils : nb de fils issu de la particule nř quel_fils
//les autres particules continuent d'évoluer
static void actualisation(PnlVect *Z, PnlVect *omega, int n_fils, int quel_fils,
{
    int i, j, k;
    PnlVect *Znew;
    double g, z;
    int nb_part_act_new;

    nb_part_act_new = somme(omega);
    Znew = pnl_vect_create(nb_part_act_new);
    //printf("nb part actives =%d \ n",nb_part_act);
    k = 0;
    for (i = 0; i < Z->size; i++)
    {
        z = GET(Z, i);
        if (i == quel_fils)
        {
            for (j = 0; j < n_fils; j++)
            {
                g = pnl_rng_normal(rng);

                LET(Znew, k) = z * exp((P->r - 0.5 * pow(P->sigma, 2)) * delta_t +
                k = k + 1;
            }
        }
        else
        {
            g = pnl_rng_normal(rng);

            LET(Znew, k) = z * exp((P->r - 0.5 * pow(P->sigma, 2)) * delta_t + P->
            k = k + 1;
        }
    }
    pnl_vect_resize(Znew, k);
    pnl_vect_clone(Z, Znew);
    pnl_vect_free(&Znew);
}

```

```
//calcul du prix par Monte Carlo
//MC : nb de tirages
//M : nombre maximal de fils possible par tirage
//a_p : vecteur des coefficients a/p
//proba : vecteur des probabilités du nombre de fils
static double PDE2(int MC, int M, PnlVect *a_p, PnlVect *proba, PnlRng *rng, con
{
    int i, l, m;
    double s, S, p, Q;
    PnlVect *Z;
    PnlVect *omega;
    int nb_part_act, quel_fils, n_fils;
    double t_courant;
    Z = pnl_vect_create(1);
    omega = pnl_vect_create(M + 1);
    S = 0;
    for (i = 0; i < MC; i++)
    {
        pnl_vect_resize(Z, 1);
        pnl_vect_set_zero(omega);
        s = pnl_rng_exp(P->beta, rng);
        if (s > P->T)
        {
            //si la particule vit plus longtemps que T
            init(Z, P->T, rng, P);
            S = S + psi(GET(Z, 0), P);
        }
        else
        {
            //si la particule meurt avant T
            t_courant = s;
            init(Z, s, rng, P);

            while (t_courant < P->T)
            {

                nb_part_act = somme(omega);
                quel_fils = tirage_uni(nb_part_act, rng); //on tire quel fils meurt
                n_fils = tirage_proba_p(proba, rng); //on tire son nb de fils
```

```

        LET(omega, n_fils) += 1;
        s = pnl_rng_exp(somme(omega) * P->beta, rng); //on tire le 1er ins
        //où une particule meurt
        actualisation(Z, omega, n_fils, quel_fils, rng, MIN(P->T - t_coura
        t_courant += s;
    }
    if (Z->size > 0)
    {
        p = 1;

        for (m = 0; m < omega->size; m++)
        {
            p *= pnl_pow_i(GET(a_p, m), (int) GET(omega, m));
        }
        Q = 1;
        for (l = 0; l < Z->size; l++)
        {
            Q *= psi(GET(Z, l), P);
        }
        S += Q * p;
    }
}

pnl_vect_free(&omega);
pnl_vect_free(&Z);
return S / (MC);
}

//calcul du prix par Monte Carlo
//MC : nb de tirages
//M : nombre maximal de fils possible par tirage
//a_p : vecteur des coefficients a/p
//proba : vecteur des probabilités du nombre de fils
static double PDE1(int MC, int M, PnlVect *a_p, PnlVect *proba, PnlRng *rng, con
{
    int i, l;
    double s, S, p, Q;
    PnlVect *Z;
    PnlVect *omega;
    int n_fils;
    double t_courant;

```

```

Z = pnl_vect_create(1);
omega = pnl_vect_create(M + 1);
S = 0;
for (i = 0; i < MC; i++)
{
    pnl_vect_resize(Z, 1);
    pnl_vect_set_zero(omega);
    s = pnl_rng_exp(P->beta / (1.0 - P->Recovery), rng);
    if (s > P->T)
    {
        //si la particule vit plus longtemps que T
        init(Z, P->T, rng, P);
        S = S + psi(GET(Z, 0), P);
    }
    else
    {
        //si la particule meurt avant T
        t_courant = s;
        init(Z, s, rng, P);
        n_fils = tirage_proba_p(proba, rng);
        LET(omega, n_fils) += 1;
        actualisation(Z, omega, n_fils, 0, rng, P->T - t_courant, P);

        if (Z->size > 0)
        {
            if (n_fils == 1) p = GET(a_p, 1) * (1.0 - P->Recovery) + P->Recovery;
            else p = GET(a_p, n_fils) * (1.0 - P->Recovery);
            //printf("taille de v %d \ n",v->size);

            Q = 1;
            for (l = 0; l < Z->size; l++)
            {
                Q *= psi(GET(Z, l), P);
            }
            S += Q * p;
        }
    }
}
pnl_vect_free(&omega);
pnl_vect_free(&Z);
//printf("la variance est de %f \ n", V/MC-(S/MC)*(S/MC));

```

```

    return S / (MC);
}

```

```

static int CALC(MC_Branch_HLabordere)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    Product P;
    PnlRng *rng;
    PnlVect *a, *p, *a_p;
    int N, MC, M;

    P.T          = ptOpt->Maturity.Val.V_DATE;
    P.K          = ptOpt->PayOff.Val.V_NUMFUNC_1->Par[0].Val.V_PDOUBLE;
    P.beta       = ptMod->Intensity.Val.V_PDOUBLE;
    P.sigma      = ptMod->Sigma.Val.V_PDOUBLE;
    P.S0         = ptMod->S0.Val.V_PDOUBLE;
    P.r          = ptMod->Interest.Val.V_PDOUBLE;
    P.Recovery   = ptMod->Recovery.Val.V_DOUBLE;
    rng          = pnl_rng_create(Met->Par[0].Val.V_ENUM.value);
    N            = Met->Par[1].Val.V_PINT;
    MC           = pnl_pow_i(2, N);
    M            = 4;
    a            = pnl_vect_create_from_zero(M + 1);
    p            = pnl_vect_create_from_zero(M + 1);
    a_p          = pnl_vect_create_from_zero(M + 1);

    LET(a, 0) = 0.0589;
    LET(a, 1) = 0.5;
    LET(a, 2) = 0.8164;
    LET(a, 3) = 0.0;
    LET(a, 4) = -0.4043;

    pnl_rng_sseed(rng, 0);
    proba(p, a);
    if (ptMod->Counterparty.Val.V_ENUM.value == 0)
    {
        coeff_PDE2(a_p, a, p);
        Met->Res[0].Val.V_PDOUBLE = PDE2(MC, M, a_p, p, rng, &P);
    }
    else

```



```

    {
        coeff_PDE1(a_p, a, p);
        Met->Res[0].Val.V_PDOUBLE = PDE1(MC, M, a_p, p, rng, &P);
    }

    pnl_vect_free(&a);
    pnl_vect_free(&p);
    pnl_vect_free(&a_p);
    pnl_rng_free(&rng);
    return OK;
}

static int CHK_OPT(MC_Branch_HLabordere)(void *Opt, void *Mod)
{
    if (strcmp(((Option *)Opt)->Name, "CVA") != 0) return FAIL;
    return OK;
}

#endif

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->HelpFilenameHint = "mc_branch_hlabordere";
        Met->Par[0].Val.V_ENUM.value = 0;
        Met->Par[0].Val.V_ENUM.members = &PremiaEnumRNGs;
        Met->Par[1].Val.V_PINT = 22;
    }
    return OK;
}

PricingMethod MET(MC_Branch_HLabordere) =
{
    "MC_Branch_HLabordere",
    {
        {"RandomGenerator", ENUM, {100}, ALLOW},
        {"N samples 2^M", PINT, {22}, ALLOW},
    }
}

```

```
    {" ", PREMIA_NULLTYPE, {0}, FORBID}  
  },  
  CALC(MC_Branch_HLabordere),  
  {  
    {"CVA", DOUBLE, {0}, FORBID},  
    {" ", PREMIA_NULLTYPE, {0}, FORBID}  
  },  
  CHK_OPT(MC_Branch_HLabordere),  
  CHK_ok,  
  MET(Init)  
};
```