

## [Help](#)

```
/* Monte Carlo Simulation for double Barrier option :
   The program provides estimations for Price and Delta with
   a confidence interval. */
/* Quasi Monte Carlo simulation is not yet allowed for this routine */

#include "
href../../mod/bs1d/bs1d_doublim/bs1d_doublim_h_src.pdfbs1d_doublim.h"
#include "
href../../common/enums_h_src.pdfenums.h"

static int proba_barrierout(double lnspot, double lastlnspot,
                           double lastlow, double lastup, double low, double up,
                           double rap, double *proba)
{
    if ((lnspot + lastlnspot) < (lastup + lastlow))
        *proba = exp(-2.*rap * ((lastlnspot - lastlow) * (lnspot - lastlow) - (lastlnspot - lastup) * (lnspot - lastup)));
    else
        *proba = exp(-2.*rap * ((lastlnspot - lastup) * (lnspot - lastup) - (lastlnspot - lastlow) * (lnspot - lastlow)));
    return OK;
}

static int MC_OutBaldi_97(NumFunc_1 *L, NumFunc_1 *U, NumFunc_1 *Rebate, double t, double M,
                           double lnspot, double lastlnspot, double price_sample, double delta_sample,
                           double lnspot_increment, double lastlnspot_increment, double price_sample_increment, double delta_sample_increment,
                           double rloc, double sigmaloc, double up, double low, double lastup, double lastlow, double proba, double proba_increment, double delta_increment,
                           double mean_price, double var_price, double mean_delta, double var_delta)
{
    double h = t / (double)M;
    double time, lnspot, lastlnspot, price_sample = 0., delta_sample;
    double lnspot_increment = 0., lastlnspot_increment, price_sample_increment = 0., delta_sample_increment;
    double rloc, sigmaloc, up, low, lastup, lastlow, proba, proba_increment, delta_increment;
    double mean_price, var_price, mean_delta, var_delta;
    long i;
    int k, inside, inside_increment;
    int init_mc;
    int simulation_dim;
    double alpha, z_alpha;

    /* Value to construct the confidence interval */
    alpha = (1. - confidence) / 2.;
    z_alpha = pnl_inv_cdfnor(1. - alpha);
```

```

/*Initialisation*/
mean_price = 0.0;
mean_delta = 0.0;
var_price = 0.0;
var_delta = 0.0;
/* Maximum Size of the random vector we need in the simulation */
simulation_dim = M;

rloc = (r - divid - SQR(sigma) / 2.) * h;
sigmaloc = sigma * sqrt(h);

/*Coefficient for the computation of the exit probability*/
rap = 1. / (sigmaloc * sigmaloc);

/*MC sampling*/
init_mc = pnl_rand_init(generator, simulation_dim, Nb);
if (init_mc == OK)
{

    for (i = 1; i <= Nb; i++)
    {
        time = 0.;
        lnspot = log(s);

        /*Up and Down Barrier at time*/
        up = log((U->Compute)(U->Par, time));
        low = log((L->Compute)(L->Par, time));

        /*Inside=0 if the path reaches the barriers*/
        inside = 1;
        inside_increment = 1;
        k = 0;

        /*Simulation of i-th path until its exit if it does*/
        while (((inside) && (k < M)) || ((inside_increment) && (k < M)))
        {
            lastlnspot = lnspot;
            lastup = up;
            lastlow = low;

```

```

time += h;
g = pnl_rand_normal(generator);
lnspot += rloc + sigmaloc * g;

lnspot_increment = lnspot + increment;
lastlnspot_increment = lastlnspot + increment;

up = log((U->Compute)(U->Par, time));
low = log((L->Compute)(L->Par, time));

/*Check if the i-th path has reached the barriers at time*/
if (inside)
    if ((lnspot > up) || (lnspot < low))
    {
        inside = 0;
        price_sample = exp(-r * time) * (Rebate->Compute)(Rebate->Pa

    }

if (inside_increment)
    if ((lnspot_increment > up) || (lnspot_increment < low))
    {
        inside_increment = 0;
        price_sample_increment = exp(-r * time) * (Rebate->Compute)(

    }

/*Check if the i-th path has reached the barriers during (time-1,t
if ((inside) && (inside_increment))
{
    proba_barrierout(lnspot, lastlnspot, lastlow, lastup, low, up,
    proba_barrierout(lnspot_increment, lastlnspot_increment, lastl
    uniform = pnl_rand_uni(generator);
    if (uniform < proba)
    {
        inside = 0;
        price_sample = exp(-r * time) * (Rebate->Compute)(Rebate->

    }
    if (uniform < proba_increment)
    {
        inside_increment = 0;
        price_sample_increment = exp(-r * time) * (Rebate->Compute
    }
}

```

```

    }

    if ((inside) && (!inside_increment))
    {
        proba_barrierout(lnspot, lastlnspot, lastlow, lastup, low, up,
        if (pnl_rand_bernoulli(proba, generator))
        {
            inside = 0;
            price_sample = exp(-r * time) * (Rebate->Compute)(Rebate->
        }
    }

    if ((!inside) && (inside_increment))
    {
        proba_barrierout(lnspot_increment, lastlnspot_increment, lastl
        if (pnl_rand_bernoulli(proba_increment, generator))
        {
            inside_increment = 0;
            price_sample_increment = exp(-r * time) * (Rebate->Compute
        }
    }
    k++;
}

if (inside)
    price_sample = exp(-r * t) * (PayOff->Compute)(PayOff->Par, exp(lnsp
if (inside_increment)
    price_sample_increment = exp(-r * t) * (PayOff->Compute)(PayOff->Par

/*Delta*/
delta_sample= (price_sample_increment - price_sample) / (increment *

/*Sum*/
mean_price += price_sample;
mean_delta += delta_sample;

/*Sum of Squares*/
var_price += SQR(price_sample);
var_delta += SQR(delta_sample);
}

```

```

    /*Price*/
    *ptprice = mean_price / (double)Nb;
    *pterror_price = sqrt(var_price / (double)Nb - SQR(*ptprice)) / sqrt(Nb - 1);
    /*Delta*/
    *ptdelta = mean_delta / (double) Nb;
    *pterror_delta = sqrt(var_delta / (double)Nb - SQR(*ptdelta)) / sqrt((double)Nb - 1);

    /* Price Confidence Interval */
    *inf_price = *ptprice - z_alpha * (*pterror_price);
    *sup_price = *ptprice + z_alpha * (*pterror_price);

    /* Delta Confidence Interval */
    *inf_delta = *ptdelta - z_alpha * (*pterror_delta);
    *sup_delta = *ptdelta + z_alpha * (*pterror_delta);
}
return init_mc;
}

```

```

int CALC(MC_OutBaldi)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r, divid;

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

    return MC_OutBaldi_97(ptOpt->LowerLimit.Val.V_NUMFUNC_1,
        ptOpt->UpperLimit.Val.V_NUMFUNC_1,
        ptOpt->Rebate.Val.V_NUMFUNC_1, ptMod->SO.Val.V_PDOUBLE,
        ptOpt->PayOff.Val.V_NUMFUNC_1,

        ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
        r,
        divid,
        ptMod->Sigma.Val.V_PDOUBLE,
        Met->Par[1].Val.V_ENUM.value,

```

```

        Met->Par[0].Val.V_LONG,
        Met->Par[2].Val.V_INT,
        Met->Par[3].Val.V_PDOUBLE,
        Met->Par[4].Val.V_PDOUBLE,
        &(Met->Res[0].Val.V_DOUBLE),
        &(Met->Res[1].Val.V_DOUBLE),
        &(Met->Res[2].Val.V_DOUBLE),
        &(Met->Res[3].Val.V_DOUBLE),
        &(Met->Res[4].Val.V_DOUBLE),
        &(Met->Res[5].Val.V_DOUBLE),
        &(Met->Res[6].Val.V_DOUBLE),
        &(Met->Res[7].Val.V_DOUBLE));
}

```

```

static int CHK_OPT(MC_OutBaldi)(void *Opt, void *Mod)
{
    Option *ptOpt = (Option *)Opt;
    TYPEOPT *opt = (TYPEOPT *) (ptOpt->TypeOpt);

    if ((opt->TwoDoubleStep).Val.V_BOOL == FALSE)
        if ((opt->Parisian).Val.V_BOOL == FALSE)
            if (((opt->OutOrIn).Val.V_BOOL == OUT) && ((opt->EuOrAm).Val.V_BOOL == EUR))
                return OK;

    return WRONG;
}

```

```

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    int type_generator;
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->Par[0].Val.V_LONG = 10000;
        Met->Par[1].Val.V_ENUM.value = 0;
        Met->Par[1].Val.V_ENUM.members = &PremiaEnumRNGs;
        Met->Par[2].Val.V_INT2 = 250;
    }
}

```

```

        Met->Par[3].Val.V_PDOUBLE = 0.01;
        Met->Par[4].Val.V_PDOUBLE = 0.95;
    }

    type_generator = Met->Par[1].Val.V_ENUM.value;

    if (pnl_rand_or_quasi(type_generator) == PNL_QMC)
    {
        Met->Res[2].Viter = IRRELEVANT;
        Met->Res[3].Viter = IRRELEVANT;
        Met->Res[4].Viter = IRRELEVANT;
        Met->Res[5].Viter = IRRELEVANT;
        Met->Res[6].Viter = IRRELEVANT;
        Met->Res[7].Viter = IRRELEVANT;

    }
    else
    {
        Met->Res[2].Viter = ALLOW;
        Met->Res[3].Viter = ALLOW;
        Met->Res[4].Viter = ALLOW;
        Met->Res[5].Viter = ALLOW;
        Met->Res[6].Viter = ALLOW;
        Met->Res[7].Viter = ALLOW;
    }
    return OK;
}

```

```

PricingMethod MET(MC_OutBaldi) =
{
    "MC_OutBaldi",
    { {"N iterations", LONG, {100}, ALLOW},
      {"RandomGenerator", ENUM, {100}, ALLOW},
      {"TimeStepNumber M", INT2, {100}, ALLOW},
      {"Delta Increment Rel", PDOUBLE, {100}, ALLOW},
      {"Confidence Value", DOUBLE, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_OutBaldi),
    { {"Price", DOUBLE, {100}, FORBID},

```

```

    {"Delta", DOUBLE, {100}, FORBID} ,
    {"Error Price", DOUBLE, {100}, FORBID},
    {"Error Delta", DOUBLE, {100}, FORBID} ,
    {"Inf Price", DOUBLE, {100}, FORBID},
    {"Sup Price", DOUBLE, {100}, FORBID} ,
    {"Inf Delta", DOUBLE, {100}, FORBID},
    {"Sup Delta", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
},
CHK_OPT(MC_OutBaldi),
CHK_mc,
MET(Init)
} ;

```