

## [Help](#)

```
/* Methods of Rogers for American Put*/

#include <stdlib.h>
#include "
href../../../../mod/bs1d/bs1d_std/bs1d_std_h_src.pdfbs1d_std.h"
#include "
href../../../../common/enums_h_src.pdfenums.h"

/*Computation of the sup over one path*/
static double sup1(double sigma, double r, double d, double K, double So, double
{
    int j;
    double t_avt, t;
    double St_avt, St;
    double Zt, Mt, test, sup;
    double put_price1, put_delta1, put_price2, put_delta2;

    Mt = 0.;
    sup = MAX(K - So, 0.) - lambda * Mt;
    test = 0.;

    for (j = 1; j <= n; j++)
    {
        t_avt = (double)(j - 1) * T / (double)n;
        t = (double)j * T / (double)n;
        St_avt = cours[j - 1];
        St = cours[j];
        Zt = exp(-r * t) * MAX(K - St, 0);
        test = (test == 1 || St_avt < K) ? 1. : 0.;
        pnl_cf_put_bs(St, K, T - t, r, d, sigma, &put_price1, &put_delta1);
        pnl_cf_put_bs(St_avt, K, T - t_avt, r, d, sigma, &put_price2, &put_delta2);
        Mt = Mt + test * (exp(-r * t) * put_price1 - exp(-r * t_avt) * put_price2);
        sup = MAX(sup, Zt - lambda * Mt);
    }

    return sup;
}
```

```

static double fv(double sigma, double r, double d, double K, double So, double T
{
    int i;
    double stockMC = 0.;
    /* cours= malloc((Np)*sizeof(double *));
    * for(i=0;i<Np;i++)
    *   cours[i] = malloc((n+1)*sizeof(double)); */
    for (i = 0; i < Np; i++)
        stockMC += sup1(sigma, r, d, K, So, T, n, cours[i], lambda);

    return stockMC / (double)Np;
}

/*Computation of lambda hat*/
static double minima(int generator, double sigma, double r, double d, double K,
{
    int i, j;
    double g, acc, diff, der;
    double la_avt, la, la_ap;
    double **path;
    double mu = r - d - sigma * sigma / 2.;

    path = malloc(Np * sizeof(double *));
    for (i = 0; i < Np; i++)
        path[i] = malloc((n + 1) * sizeof(double));
    for (i = 0; i < Np; i++)
    {
        path[i][0] = So;
        for (j = 1; j < n + 1; j++)
        {
            g = pnl_rand_normal(generator);
            path[i][j] = path[i][j - 1] * exp(sigma * sqrt(T / (double)n) * g + mu
        }
    }

    acc = 0.000000001;
    la_avt = -30.;
    la = 0.;
    la_ap = 30.;
    diff = la_ap - la_avt;

```

```

while (fabs(diff) >= 0.001)
{
    der = (fv(sigma, r, d, K, So, T, n, Np, path, la + acc) - fv(sigma, r, d,
    if (der <= 0.)
    {
        la_avt = la;
        la = (la + la_ap) / 2.;
    }
    else
    {
        la_ap = la;
        la = (la_avt + la) / 2.;
    }
    diff = la_ap - la_avt;
}

for (i = 0; i < Np; i++)
    free(path[i]);
free(path);
return la;
}

static int MC_Rogers(double So, NumFunc_1 *p, double T, double r, double divid,
{
    double mean_price, mean_delta, var_price, var_delta, price_sample, delta_sampl

    int init_mc;
    int simulation_dim = 1;
    double alpha, z_alpha;
    int i, j, m;
    double h;
    double t, t_avt, Wt, Wt_avt;
    double St, St_avt, Zt, Mt;
    double Sth, St_avth, Zth, Mth;
    double sup, suph;
    double *Sn;
    double *gn;
    double lambda, lambdah;
    double test, testh;
    double K, mu;
    double put_price1, put_delta1, put_price2, put_delta2;

```

```

Sn = malloc((n + 1) * sizeof(double));
gn = malloc((n + 1) * sizeof(double));

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

/*MC sampling*/
/* Test after initialization for the generator */
init_mc = pnl_rand_init(generator, simulation_dim, N);

if (init_mc == OK)
{
    /*Initialisation*/
    mu = r - divid - sigma * sigma / 2.;
    K = p->Par[0].Val.V_DOUBLE;
    lambda = minima(generator, sigma, r, divid, K, So, T, n, Np);
    lambdah = lambda;
    h = inc;
    mean_price = 0.0;
    mean_delta = 0.0;
    var_price = 0.0;
    var_delta = 0.0;

    /* Begin N iterations */
    for (i = 0; i <= N - 1 ; i++)
    {
        /*Simulation of one path*/
        for (m = 0; m <= n; m++)
            gn[m] = pnl_rand_normal(generator);

        Sn[0] = 0;
        for (m = 1; m <= n; m++)
            Sn[m] = Sn[m - 1] + gn[m - 1];

        /*Computation of the sup over the path*/
        Mt = 0.;
        Mth = 0.;
        sup = MAX(K - So, 0) - lambda * Mt;
    }
}

```

```

suph = MAX(K - (So + h), 0) - lambdah * Mth;
test = 0.;
testh = 0.;

for (j = 1; j <= n; j++)
{
    t_avt = (double)(j - 1) * T / (double)n;
    t = (double)j * T / (double)n;
    Wt_avt = sqrt(T / (double)n) * Sn[j - 1];
    Wt = sqrt(T / (double)n) * Sn[j];

    /*Computation of Yo(So)*/
    St_avt = So * exp(sigma * Wt_avt + mu * t_avt);
    St = So * exp(sigma * Wt + mu * t);
    Zt = exp(-r * t) * MAX(K - St, 0);
    test = (test == 1 || St_avt < K) ? 1. : 0.;
    pnl_cf_put_bs(St, K, T - t, r, divid, sigma, &put_price1, &put_del
    pnl_cf_put_bs(St_avt, K, T - t_avt, r, divid, sigma, &put_price2,

    Mt += test * (exp(-r * t) * put_price1 - exp(-r * t_avt) * put_pri
    sup = MAX(sup, Zt - lambda * Mt);

    /*Computation of Yo(So+h)*/
    St_avth = (So + h) * exp(sigma * Wt_avt + mu * t_avt);
    Sth = (So + h) * exp(sigma * Wt + mu * t);
    Zth = exp(-r * t) * MAX(K - Sth, 0);
    testh = (testh == 1 || St_avth < K) ? 1. : 0.;
    pnl_cf_put_bs(Sth, K, T - t, r, divid, sigma, &put_price1, &put_de
    pnl_cf_put_bs(St_avth, K, T - t_avt, r, divid, sigma, &put_price2,

    Mth += testh * (exp(-r * t) * put_price1 - exp(-r * t_avt) * put_p
    suph = MAX(suph, Zth - lambdah * Mth);
}

/*Sum*/
price_sample = sup;
delta_sample = (suph - sup) / h;

mean_price += price_sample;
mean_delta += delta_sample;

```

```

        /*Sum of squares*/
        var_price += SQR(price_sample);
        var_delta += SQR(delta_sample);
    }
    /* End N iterations */

    /* Price */
    *ptprice = mean_price / (double) N;
    *pterror_price = sqrt(var_price / (double)N - SQR(*ptprice)) / sqrt(N - 1)

    /*Delta*/
    *ptdelta = mean_delta / (double) N;
    *pterror_delta = sqrt(var_delta / (double)N - SQR(*ptdelta)) / sqrt((double)N - 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);
}
free(Sn);
free(gn);
return init_mc;
}

```

```

int CALC(MC_Rogers)(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_Rogers(ptMod->S0.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[0].Val.V_LONG,
        Met->Par[1].Val.V_INT,
        Met->Par[2].Val.V_INT,
        Met->Par[3].Val.V_ENUM.value,
        Met->Par[4].Val.V_PDOUBLE,
        Met->Par[5].Val.V_DOUBLE,
        &(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_Rogers)(void *Opt, void *Mod)
{
    /* Option* ptOpt=(Option*)Opt;
       TYPEOPT* opt=(TYPEOPT*)(ptOpt->TypeOpt);*/
    if ((strcmp(((Option *)Opt)->Name, "PutAmer") == 0))
        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 = 30000;
    Met->Par[1].Val.V_INT = 40;
    Met->Par[2].Val.V_INT = 300;
    Met->Par[3].Val.V_ENUM.value = 0;
    Met->Par[3].Val.V_ENUM.members = &PremiaEnumRNGs;
    Met->Par[4].Val.V_PDOUBLE = 0.01;
    Met->Par[5].Val.V_PDOUBLE = 0.95;

}
type_generator = Met->Par[3].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_Rogers) =
{
    "MC_Rogers",
    { {"N iterations", LONG, {100}, ALLOW},
      {"Time step number", INT, {100}, ALLOW},
      {"N iterations Minimisation ", INT, {100}, ALLOW},
    }
}

```



```

    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Delta Increment Rel (Digit)", PDOUBLE, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
},
CALC(MC_Rogers),
{ {"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_Rogers),
CHK_mc,
MET(Init)
};

```