

[Help](#)

```
#include "mer1d_std.h"
#include "pnl/pnl_cdf.h"

static int MCMerton(double s, NumFunc_1 *p, double t, double r, double divid, d
{
    long i;
    double mean_price, mean_delta, var_price, var_delta, forward, forward_stock, p
    double g;
    int init_mc;
    int simulation_dim = 1;
    double alpha, z_alpha;
    /* double eps=1.0;*/

    long nj, j;
    double poisson_jump, mm, Eu;

    Eu = exp(mu + 0.5 * gamma2) - 1.;
    mm = r - divid - lambda * Eu;
    /* Value to construct the confidence interval */
    alpha = (1. - confidence) / 2.;
    z_alpha = pnl_inv_cdfnor(1. - alpha);

    /*Initialisation*/
    s_plus = s * (1. + inc);
    s_minus = s * (1. - inc);
    mean_price = 0.0;
    mean_delta = 0.0;
    var_price = 0.0;
    var_delta = 0.0;

    /*Median forward stock and delta values*/
    sigma_sqrt = sigma * sqrt(t);
    forward = exp((mm - SQR(sigma) / 2.0) * t);
    forward_stock = s * forward;

    /*MC sampling*/
    init_mc = pnl_rand_init(generator, simulation_dim, N);
```

```

/* Test after initialization for the generator */
if (init_mc == OK)
{

    /* Begin N iterations */
    for (i = 1 ; i <= N ; i++)
    {
        /* Simulation of a gaussian variable according to the generator type,
           that is Monte Carlo or Quasi Monte Carlo. */
        g = pnl_rand_normal(generator);
        exp_sigmaxwt = exp(sigma_sqrt * g);

        /* Jump */
        nj = pnl_rand_poisson(lambda * t, generator);
        poisson_jump = 1.;
        for (j = 1; j <= nj; j++)
        {
            g = pnl_rand_normal(generator);
            poisson_jump *= (exp(mu + sqrt(gamma2) * g));
        }

        S_T = forward_stock * exp_sigmaxwt * poisson_jump;
        U_T = forward * exp_sigmaxwt * poisson_jump;

        /*Price*/
        price_sample = (p->Compute)(p->Par, S_T);

        /*Delta*/
        price_sample_plus = (p->Compute)(p->Par, U_T * s_plus);
        price_sample_minus = (p->Compute)(p->Par, U_T * s_minus);
        delta_sample = (price_sample_plus - price_sample_minus) / (2.*s * inc)
        /*Sum*/
        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 = exp(-r * t) * (mean_price / (double) N);
*pterror_price = sqrt(exp(-2.0 * r * t) * var_price / (double)N - SQR(*ptprice));

/*Delta*/
*ptdelta = exp(-r * t) * mean_delta / (double) N;
*pterror_delta = sqrt(exp(-2.0 * r * t) * (var_delta / (double)N - SQR(*ptdelta)));

/* 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_Merton)(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 MCMerton(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,
                    ptMod->Lambda.Val.V_PDOUBLE,
                    ptMod->Mean.Val.V_PDOUBLE,
                    ptMod->Variance.Val.V_PDOUBLE,
```

```

        Met->Par[0].Val.V_LONG,
        Met->Par[1].Val.V_ENUM.value,
        Met->Par[2].Val.V_PDOUBLE,
        Met->Par[3].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_Merton)(void *Opt, void *Mod)
{
    Option *ptOpt = (Option *)Opt;
    TYPEOPT *opt = (TYPEOPT *) (ptOpt->TypeOpt);

    if ((opt->EuOrAm).Val.V_BOOL == EURO)
        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 = 50000;
        Met->Par[1].Val.V_ENUM.value = 0;
        Met->Par[1].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[2].Val.V_PDOUBLE = 0.01;
        Met->Par[3].Val.V_DOUBLE = 0.95;
        Met->Par[4].Val.V_PDOUBLE = 0.1;
        Met->Par[5].Val.V_DOUBLE = 0.0;
        Met->Par[6].Val.V_DOUBLE = 0.16;
    }
}

```

```

    }

    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_Merton) =
{
    "MC_Merton",
    { {"N iterations", LONG, {100}, ALLOW},
      {"RandomGenerator", ENUM, {100}, ALLOW},
      {"Delta Increment Rel (Digit)", PDOUBLE, {100}, ALLOW},
      {"Confidence Value", DOUBLE, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_Merton),
    { {"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_Merton),
CHK_mc,
MET(Init)
};
```