

## [Help](#)

```
/* Standard Monte Carlo simulation for a Call - Put - Floating Asian option.  
   In the case of Monte Carlo simulation, the program provides estimations for p  
   In the case of Quasi-Monte Carlo simulation, the program just provides estima
```

```
#include "  
href../../mod/bs1d/bs1d_pad/bs1d_pad_h_src.pdfbs1d_pad.h"  
#include "  
href../../common/enums_h_src.pdfenums.h"  
  
/* Simulation of the final spot and the average with one of the three schemes */  
static void Simul_StockAndAverage(int scheme, int generator, int step_number, do  
{  
    double g1, g2, integral, bb, w_t, w_t_1, S_t = 0., current_t;  
    double h = T / step_number;  
    double sqrt_h = sqrt(h);  
    double trend = (r - divid) - 0.5 * SQR(sigma);  
    int i;  
  
    /*Initialisation*/  
    w_t = 0.0;  
    current_t = 0.0;  
    integral = 0.;  
  
    /*Average and Stock Computation*/  
  
    /* Scheme 1 : Rieman sums */  
    if ((scheme != 2) && (scheme != 3))  
    {  
        /* Simulation of M gaussian variables according to the generator type,  
           that is Monte Carlo or Quasi Monte Carlo. */  
        g1 = pnl_rand_gauss(step_number, CREATE, 0, generator);  
  
        for (i = 0; i < step_number; i++)  
        {  
            S_t = x * exp(trend * current_t + sigma * w_t);  
            integral += S_t;
```

```

        current_t += h;
        /* gaussian value from the table Gaussians */
        g1 = pnl_rand_gauss(step_number, RETRIEVE, i, generator);
        w_t += sqrt_h * g1;
    }
}
else
{
    /* Scheme 2 : Trapezoidal method */
    if (scheme == 2)
    {
        /* Simulation of M gaussian variables according to the generator type,
        that is Monte Carlo or Quasi Monte Carlo. */
        g1 = pnl_rand_gauss(step_number, CREATE, 0, generator);

        for (i = 0; i < step_number; i++)
        {
            /* gaussian value from the table Gaussians */
            g1 = pnl_rand_gauss(step_number, RETRIEVE, i, generator);
            /*printf("g= %.3lf\ n", g1);*/
            w_t_1 = sqrt_h * g1 + w_t;

            S_t = x * exp(trend * current_t + sigma * w_t);
            integral += S_t * (1 + (r - divid) * h / 2. + sigma * (w_t_1 - w_t));

            current_t += h;
            w_t = w_t_1;
        }
        /*printf("\ n");*/
    }
    else
    {
        /* Scheme 3 : Brownian Bridge method */

        /* Simulation of 2M gaussian variables according to the generator type
        that is Monte Carlo or Quasi Monte Carlo. */
        g1 = pnl_rand_gauss(2 * step_number, CREATE, 0, generator);

        for (i = 0; i < step_number; i++)
        {
            g1 = pnl_rand_gauss(step_number, RETRIEVE, 2 * i, generator);

```

```

        w_t_1 = sqrt_h * g1 + w_t;

        g2 = pnl_rand_gauss(step_number, RETRIEVE, (2 * i) + 1, generator);
        bb = (w_t + w_t_1) / 2. + g2 * sqrt(h / 6.);

        S_t = x * exp(trend * current_t + sigma * w_t);
        integral += S_t * (1 + (r - divid) * h / 2. + sigma * (bb - w_t));

        current_t += h;
        w_t = w_t_1;
    }
}

/*Stock*/
*ptstock = S_t;

/*Average*/
*ptaverage = integral / step_number;

return;
}

static int FloatingAsian_StandardMC(double s, double time_spent, double pseudo_
{
    long i;
    int init_mc;
    int simulation_dim;
    double price_sample, delta_sample, mean_price, mean_delta, var_price, var_delt
    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;
/* Size of the random vector we need in the simulation */
if (scheme == 3)
    simulation_dim = 2 * M;
else
    simulation_dim = M;

/*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++)
    {
        Simul_StockAndAverage(scheme, generator, M, t, s, r, divid, sigma, &St

        /*Price*/
        price_sample = (p->Compute)(p->Par, St, pseudo_strike + average * (1.
        /*price_inc1 = (p->Compute)(p->Par, (1.+inc)*St, pseudo_strike+(1.+inc
        price_inc2 = (p->Compute)(p->Par, (1.-inc)*St, pseudo_strike+(1.-in
        /*Delta*/
        if (price_sample > 0)
            /*delta_sample= (price_inc1 - price_inc2)/(2.*s*inc);*/
            delta_sample = (St - (1 - time_spent) * average) / s;
        else
            delta_sample = 0;
        /*delta_sample= (price_inc1 - price_inc2)/(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(*ptp
/* Price Confidence Interval */
*inf_price = *ptprice - z_alpha * (*pterror_price);
*sup_price = *ptprice + z_alpha * (*pterror_price);

/*Delta*/
*ptdelta = exp(-r * t) * mean_delta / (double) N;
/* Put Case */
if ((p->Compute) == &Put_StrikeSpot2)
    *ptdelta *= (-1);

*pterror_delta = sqrt(exp(-2.0 * r * t) * (var_delta / (double)N - SQR(*pt
/* Delta Confidence Interval */
*inf_delta = *ptdelta - z_alpha * (*pterror_delta);
*sup_delta = *ptdelta + z_alpha * (*pterror_delta);
}
return init_mc;
}

```

```

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

    double T, t_0, T_0;
    int return_value;

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

    T = ptOpt->Maturity.Val.V_DATE;
    T_0 = ptMod->T.Val.V_DATE;
    t_0 = (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE;
    time_spent = (T_0 - t_0) / (T - t_0);

```

```

if (T_0 < t_0)
{
    Fprintf(TOSCREEN, "T_0 < t_0, untreated case\ n\ n\ n");
    return_value = WRONG;
}

/* Case t_0 <= T_0 */
else
{
    pseudo_strike = time_spent * (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[4].Val.
    return_value = FloatingAsian_StandardMC(ptMod->S0.Val.V_PDOUBLE,
                                           time_spent,
                                           pseudo_strike,
                                           ptOpt->PayOff.Val.V_NUMFUNC_2,
                                           ptOpt->Maturity.Val.V_DATE - ptMod
                                           r,
                                           divid,
                                           ptMod->Sigma.Val.V_PDOUBLE,
                                           Met->Par[2].Val.V_LONG,
                                           Met->Par[0].Val.V_INT2,
                                           Met->Par[3].Val.V_ENUM.value,
                                           Met->Par[1].Val.V_ENUM.value,
                                           Met->Par[4].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));

}

return return_value;
}

static int CHK_OPT(MC_FloatingAsian_Standard)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "AsianCallFloatingEuro") == 0) || (strcmp((

```

```

    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_INT2 = 100;
        Met->Par[1].Val.V_ENUM.value = 0;
        Met->Par[1].Val.V_ENUM.members = &PremiaEnumRNGs;
        Met->Par[2].Val.V_LONG = 20000;
        Met->Par[3].Val.V_ENUM.value = 2;
        Met->Par[3].Val.V_ENUM.members = &PremiaEnumIntegralScheme;
        Met->Par[4].Val.V_DOUBLE = 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_FloatingAsian_Standard) =
{
    "MC_FloatingAsian_Standard",
    { {"TimeStepNumber", INT2, {100}, ALLOW},
      {"RandomGenerator", ENUM, {100}, ALLOW},
      {"N iterations", LONG, {100}, ALLOW},
      {"Integral Scheme:", ENUM, {100}, FORBID},
      {"Confidence Value", DOUBLE, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_FloatingAsian_Standard),
    { {"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_FloatingAsian_Standard),
    CHK_ok,
    MET(Init)
};

```