

## [Help](#)

```
/* Exact method for the computation of a Call or a Put Fixed strike Asian
   option. In the case of Monte Carlo simulation, the program provides
   estimations for price and delta with a confidence interval. The case of
   Quasi-Monte Carlo is not handled. */
#include "
href../../mod/bs1d/bs1d_pad/bs1d_pad_h_src.pdfbs1d_pad.h"
#include "
href../../common/enums_h_src.pdfenums.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
static int CHK_OPT(MC_FixedAsian_Exact)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_FixedAsian_Exact)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

/* Function which sorts a vector of real numbers
   * and keeps track of the former indices */
static void vector_sort(double *tableau, int *indice, int longueur)
{
    int i, compt, marqueur;
    double memory;
    for (i = 1; i < longueur; i++)
    {
        memory = tableau[i];
        compt = i - 1;

        do
        {
            marqueur = 0;
            if (tableau[compt] > memory)
            {
                tableau[compt + 1] = tableau[compt];
                indice[compt + 1] = indice[compt];
                compt--;
            }
        }
    }
}
```

```

        marqueur = 1;
    }
    if (compt < 0) marqueur = 0;
}
while (marqueur);
tableau[compt + 1] = memory;
indice[compt + 1] = i;
}
}

/* Function which computes the integrand in the Girsanov weight */
static double PHI(double t, double x, double gamma, double sigma, double Wfinal,
{
    if (t < 1.e-7) /* To avoid overflow problems */
        return cT * sqrt(t) + x * (2 * x * x / (3 * sigma * sigma * t) - 0.5) / sqrt(t);
    else
        return cT * sqrt(t) + (-exp(-2 * x) + exp(-x) * t * (sigma * sigma - 2 * gamma * x));
}

/* Function which returns one simulation of the unbiased estimator */
static void UnbiasedEstimator(double r, double sigma, double T, double divid, double d)
{
    double Z_T, Nalgo;
    int Npoisson, i, j, k;
    double gamma = r - divid - SQR(sigma) / 2;
    double T3 = SQR(T) * T / 3;
    double s;
    int l, m, indice;
    int *V_Npoisson = malloc(n * sizeof(int));
    double *x, *y;
    int *ind;

    Z_T = (sigma * sqrt(T / 3)) * pnl_rand_normal(generator) + gamma * T / 2;
    Npoisson = 0;
    for (l = 0; l < n; l++)
    {
        V_Npoisson[l] = pnl_rand_poisson(cP * T, generator);
        Npoisson += V_Npoisson[l];
    }

    x = malloc(Npoisson * sizeof(double));

```

```

y = malloc(Npoisson * sizeof(double));
ind = malloc(Npoisson * sizeof(int));

for (l = 0; l < Npoisson; l++)
    ind[l] = 1;
Nalgo = 1;
if (Npoisson != 0)
{
    for (j = 0; j < Npoisson; j++)
    {
        x[j] = T * pnl_rand_uni(generator);
        x[j] = x[j] * x[j] / T;
    }
    vector_sort(x, ind, Npoisson);

    y[Npoisson - 1] = (T * (Z_T - gamma * T / 2) / sigma) * (pow(x[Npoisson - 1], 3) / pow(x[Npoisson - 2], 3) + sqrt((pow(x[Npoisson - 1], 3) / pow(x[Npoisson - 2], 3) - 1)));
    for (i = Npoisson - 2; i > -1; i--)
    {
        y[i] = y[i + 1] * pow(x[i], 3) / pow(x[i + 1], 3) + sqrt((pow(x[i + 1], 3) / pow(x[i], 3) - 1));
    }
    s = 0;
    m = 0;
    for (l = 0; l < n; l++)
    {
        Nalgo = 1;
        for (k = m; k < m + V_Npoisson[l]; k++)
        {
            indice = ind[k];
            Nalgo = Nalgo * 2 * PHI(x[indice], sigma * y[indice] / x[indice] + 1) / (sigma * y[indice]);
        }
        s += Nalgo;
        m += V_Npoisson[l];
    }
    Nalgo = s / n;
}

*Z = Z_T;
*Girsanov_Weight = exp((Z_T * (Z_T / 2 - 1) + 1 - exp(-Z_T)) / (sigma * sigma));

free(x);
free(y);

```

```

    free(ind);
    free(V_Npoisson);
}

/* -----*/
/* Pricing of an asian option by the exact method
   Estimator of the price and the delta.
   s et K are pseudo-spot and pseudo-strike. */
/* ----- */
static int FixedAsian_Exact(double s, double K, double time_spent, NumFunc_2 *p
{
    long i;
    double d1, d2, N_d1, N_d2;
    double price_Q, delta_Q;
    double price_sample, delta_sample = 0., mean_price, mean_delta, var_price, var
    int init_mc;
    double alpha, z_alpha;

    /* Definition of the optimal parameters */
    const double cP = 1. / (2 * t); //parameter for the Poisson variable
    const double cT = cP; //shift parameter
    const int n = 5; //conditional sampling

    double Z, Girsanov_Weight;

    /* 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;
    r_d = r - divid;

    /* MC sampling */
    init_mc = pnl_rand_init(generator, 1, nb);

```

```

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

    /* Computation of the price and the delta for the term Q with the control
    d1 = (log(s / K) + (r_d + sigma * sigma / 6.0) * (t / 2.)) / (sigma * sqrt
    d2 = d1 - sigma * sqrt(t / 3.);

    /* Case where computing the Put is better */
    if (s <= K * exp(r * t))
    {
        const double cst_KV = 0.9;
        N_d1 = cdf_nor(-d1);
        N_d2 = cdf_nor(-d2);
        price_Q = exp(-r * t) * (K * N_d2 - s * exp((r_d - sigma * sigma / 6.0
        delta_Q = exp(-r * t) * (-exp((r_d - sigma * sigma / 6.0) * (t / 2.))
        /* Beginning of the N iterations */
        for (i = 1; i <= nb; i++)
        {
            /* Price */
            (void)UnbiaisedEstimator(r, sigma, t, divid, cP, cT, n, generator,
            delta_sample = 0.0;
            price_sample = MAX(K - s * exp(Z), 0.) * (Girsanov_Weight - cst_KV
            if (price_sample > 0.0)
                delta_sample -= exp(Z) * (Girsanov_Weight - cst_KV);

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

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

        /* Price estimator */
        *ptprice = (mean_price / (double)nb);
        *pterror_price = exp(-r * t) * sqrt(var_price / (double)nb - SQR(*ptpr
        /* Delta estimator */
        *ptdelta = (mean_delta / (double)nb);

```

```

*pterror_delta = exp(-r * t) * sqrt(var_delta / (double)nb - SQR(*ptde
/* Put case */
if ((p->Compute) == &Put_OverSpot2)
{
    *ptprice = exp(-r * t) * (*ptprice) + cst_KV * price_Q;
    *ptdelta = exp(-r * t) * (*ptdelta) + cst_KV * delta_Q;
}
/* Call case : use of Call-Put parity */
else
{
    if (r_d == 0)
    {
        *ptprice = exp(-r * t) * (*ptprice) + cst_KV * price_Q + s * e
        *ptdelta = exp(-r * t) * (*ptdelta) + cst_KV * delta_Q + exp(-
    }
    else
    {
        *ptprice = exp(-r * t) * (*ptprice) + cst_KV * price_Q + s * (
        *ptdelta = exp(-r * t) * (*ptdelta) + cst_KV * delta_Q + (exp(
    }
}
/* 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);
}
/* Case where computing the Call is better */
else
{
    const double cst_KV = 1.1;
    N_d1 = cdf_nor(d1);
    N_d2 = cdf_nor(d2);
    price_Q = exp(-r * t) * (s * exp((r_d - sigma * sigma / 6.0) * (t / 2.
    delta_Q = exp(-r * t) * exp((r_d - sigma * sigma / 6.0) * (t / 2.)) *
    /* Begin of the N iterations */
    for (i = 1; i <= nb; i++)
    {
        /* Price */
        (void)UnbiaisedEstimator(r, sigma, t, divid, cP, cT, n, generator,

```

```

    delta_sample = 0.0;
    price_sample = MAX(s * exp(Z) - K, 0.) * (Girsanov_Weight - cst_KV);
    if (price_sample > 0.0)
        delta_sample += exp(Z) * (Girsanov_Weight - cst_KV);

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

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

/* Price estimator */
*ptprice = (mean_price / (double)nb);
*pterror_price = exp(-r * t) * sqrt(var_price / (double)nb - SQR(*ptprice));
/* Delta estimator */
*ptdelta = (mean_delta / (double)nb);
*pterror_delta = exp(-r * t) * sqrt(var_delta / (double)nb - SQR(*ptdelta));
/* Call case */
if ((p->Compute) == &Call_OverSpot2)
{
    *ptprice = exp(-r * t) * (*ptprice) + cst_KV * price_Q;
    *ptdelta = exp(-r * t) * (*ptdelta) + cst_KV * delta_Q;
}
/* Put case : use of Call-Put parity */
else
{
    if (r_d == 0)
    {
        *ptprice = exp(-r * t) * (*ptprice) + cst_KV * price_Q - s * e;
        *ptdelta = exp(-r * t) * (*ptdelta) + cst_KV * delta_Q - exp(-r * t);
    }
    else
    {
        *ptprice = exp(-r * t) * (*ptprice) + cst_KV * price_Q - s * e;
        *ptdelta = exp(-r * t) * (*ptdelta) + cst_KV * delta_Q - (exp(-r * t) * price_Q);
    }
}

```

```

        /* 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_FixedAsian_Exact)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;

    double T, t_0, T_0;
    double r, divid, time_spent, pseudo_strike, true_strike, pseudo_spot;
    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_spot = (1. - time_spent) * ptMod->S0.Val.V_PDOUBLE;
        pseudo_strike = (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE - ti

```

```

true_strike = (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE;

(ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE = pseudo_strike;

if (pseudo_strike <= 0.)
{
    Fprintf(TOSCREEN, "FORMULE ANALYTIQUE\ n\ n\ n");
    return_value = Analytic_KemnaVorst(pseudo_spot,
                                        pseudo_strike,
                                        time_spent,
                                        ptOpt->PayOff.Val.V_NUMFUNC_2,
                                        T - T_0,
                                        r,
                                        divid,
                                        &(Met->Res[0].Val.V_DOUBLE),
                                        &(Met->Res[1].Val.V_DOUBLE));
}
else
    return_value = FixedAsian_Exact(pseudo_spot,
                                    pseudo_strike,
                                    time_spent,
                                    ptOpt->PayOff.Val.V_NUMFUNC_2,
                                    T - T_0,
                                    r,
                                    divid,
                                    ptMod->Sigma.Val.V_PDOUBLE,
                                    Met->Par[1].Val.V_ENUM.value,
                                    Met->Par[0].Val.V_INT,
                                    Met->Par[2].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));

(ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE = true_strike;
}

```

```

    return return_value;
}

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

        return OK;

    return WRONG;
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    /*int type_generator;*/
    if (Met->init == 0)

    {
        Met->init = 1;

        Met->Par[0].Val.V_ENUM.value = 0;
        Met->Par[0].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[1].Val.V_LONG = 20000;
        Met->Par[2].Val.V_DOUBLE = 0.95;
    }
    /*type_generator= Met->Par[0].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_FixedAsian_Exact) =
{
    "MC_FixedAsian_ExactMethod",
    { {"RandomGenerator", ENUM, {100}, ALLOW},
      {"N iterations", LONG, {100}, ALLOW},
      {"Confidence Value", DOUBLE, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_FixedAsian_Exact),
    { {"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_FixedAsian_Exact),
    CHK_ok,
    MET(Init)
};

```