

[Help](#)

```
#include "hes1d_std.h"
#include <pnl/pnl_mathtools.h>
#include <pnl/pnl_root.h>

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2012+2) //The "#els
static int CHK_OPT(AP_SmallTime_ImpliedVolatility)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(AP_SmallTime_ImpliedVolatility)(void *Opt, void *Mod, PricingMethod *Me
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

//Calculating the implied volatility under Heston model for the short et long ma
//This code is based on the works of Forde, Jacquier and Mijatovic
static double arctan(double y)
{
    double error = 0.000000000001;
    double inf = -0.5 * M_PI;
    double sup = 0.5 * M_PI;
    double res = 0.0;
    double errorMethode = 1.0;

    while ((sup - inf > error) && errorMethode > error)
    {
        res = (sup + inf) / 2.0 ;
        if (y > tan(res))
            inf = res;
        else
            sup = res;

        errorMethode = ABS(y - tan(res));
    }
    return res;
}
```

```

//The Rate function. Used for the short maturity case
static double LambdaFunc(double kappa, double theta, double sigma, double rho, d
{
    double res;
    double Cot;

    Cot = cos(0.5 * sigma * p * sqrt(1 - rho * rho)) / sin(0.5 * sigma * p * sqrt(
    res = p * v0 / (sigma * (sqrt(1 - rho * rho) * Cot - rho)) ;
    return res;
}

//The derivative of the Rate function
static double LambdaDerivee(double kappa, double theta, double sigma, double rho
{
    double res;
    double Cot;

    Cot = cos(0.5 * sigma * p * sqrt(1 - rho * rho)) / sin(0.5 * sigma * p * sqrt(
    res = v0 / (sigma * (sqrt(1 - rho * rho) * Cot - rho)) +
        p * v0 * (1 - rho * rho) / (SQR(sqrt(1 - rho * rho) * Cot - rho) * SQR(s
    return res;
}

int ApIVSmallHeston(double S0, NumFunc_1 *Payoff, double T, double r, double di
{
    double K;
    double x ;
    double p = 0.0;
    double Lambda;
    double Lambda1;
    double p_plus = 0.0, p_minus = 0.0;
    double sup, inf;
    double Error = 0.0000000001;
    double ErrorMethode = 1;

    K = Payoff->Par[0].Val.V_PDDOUBLE;
    x = log(K / S0);
    if (rho < 0)
    {
        p_minus = arctan(sqrt(1 - rho * rho) / rho) / (0.5 * sigma * sqrt(1 - rho
        p_plus = (M_PI + arctan(sqrt(1 - rho * rho) / rho)) / (0.5 * sigma * sqrt(

```

```

    }
    if (rho == 0)
    {
        p_minus = -M_PI / sigma ;
        p_plus = M_PI / sigma ;
    }

    if (rho > 0)
    {
        p_minus = (-M_PI + arctan(sqrt(1 - rho * rho) / rho)) / (0.5 * sigma * sqrt(1 - rho * rho));
        p_plus = arctan(sqrt(1 - rho * rho) / rho) / (0.5 * sigma * sqrt(1 - rho * rho));
    }

    if (x == 0)
        x = 0.00001;

    //Legendre Transform of the Rate function function
    inf = p_minus;
    sup = p_plus;
    while ((sup - inf) > Error && ErrorMethode > Error)
    {
        p = (sup + inf) / 2.0;

        Lambda1 = LambdaDerivee(kappa, theta, sigma, rho, v0, p);

        if (Lambda1 < x)
            inf = p;
        else
            sup = p;

        ErrorMethode = ABS(Lambda1 - x);
    }

    Lambda = x * p - LambdaFunc(kappa, theta, sigma, rho, v0, p);

    /* Implied Volatility*/
    *implied_vol = (ABS(x) / sqrt(2.0 * Lambda));

    return OK;

```

```

}

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

    if (ptMod->Sigma.Val.V_PDOUBLE == 0.0)
    {
        Fprintf(TOSCREEN, "BLACK-SHOLES MODEL\ n\ n\ n");
        return WRONG;
    }
    else
    {
        r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
        divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

        return ApIVSmallHeston(ptMod->S0.Val.V_PDOUBLE,
                                ptOpt->PayOff.Val.V_NUMFUNC_1,
                                ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                                r,
                                divid, ptMod->Sigma0.Val.V_PDOUBLE
                                , ptMod->MeanReversion.Val.V_PDOUBLE,
                                ptMod->LongRunVariance.Val.V_PDOUBLE,
                                ptMod->Sigma.Val.V_PDOUBLE,
                                ptMod->Rho.Val.V_PDOUBLE,
                                &(Met->Res[0].Val.V_DOUBLE)
                                );
    }
}

}

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

        return OK;
    return WRONG;
}

```

```
#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;
    }

    return OK;
}

PricingMethod MET(AP_SmallTime_ImpliedVolatility) =
{
    "AP_SmallTime_ImpliedVolatility",
    {{" ", PREMIA_NULLTYPE, {0}, FORBID}},
    CALC(AP_SmallTime_ImpliedVolatility),
    { {"Implied Volatility for Small-Time", DOUBLE, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(AP_SmallTime_ImpliedVolatility),
    CHK_ok,
    MET(Init)
};
```