

[Help](#)

```
extern "C" {
#include "bs1d_pad.h"
#include "math/linsys.h"
#include "pnl/pnl_cdf.h"
}
#include <cmath>
#include <limits>
#include <iostream>

using namespace std;

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
#else

// Brackets a root of function f between x1 and x2 :
static void zerenc(double f(const double), double &x1, double &x2)
{
    const int Imax = 100; // Maximum number of iterations allowed
    const double PHI = 1.6; // Interval magnification constant
    double f1 = 0, f2 = 0;

    if (x1 == x2)
        cerr << "zerenc : irrelevant initial range" << endl;
    f1 = f(x1);
    f2 = f(x2);
    for (int i = 0; i < Imax; i++)
    {
        if (f1 * f2 < 0.0) // A root was found
            return;
        if (fabs(f1) < fabs(f2)) // x1 nearer to root : to be moved
        {
            x1 += PHI * (x1 - x2);
            f1 = f(x1);
        }
        else // x2 nearer to root : to be moved
        {
            x2 += PHI * (x2 - x1);
            f2 = f(x2);
        }
    }
}
```

```
    }
    cerr << "zerenc : too many iterations" << endl;
    return;
}

// Finds a root of function f by bisection method.
// First uses zerenc to bracket the root :
static double zerobisec(double f(const double))
{
    const int Imax = 100; // Maximum number of iterations allowed
    const double TOL = 1e-10; // Demanded precision
    double dx, fx, fmid, xmid, zero, x1 = -1, x2 = 1;

    zerenc(f, x1, x2); // First brackets function f's root
    fx = f(x1);

    if (fx < 0.0) // Then f(x2) > 0
    {
        dx = x2 - x1; // Bracketing interval
        zero = x1; // Zero is set as the bracketing interval's lower bound
    }
    else
    {
        dx = x1 - x2;
        zero = x2;
    }

    for (int i = 0; i < Imax; i++)
    {
        dx *= 0.5; // Bisection method : divides the interval into two parts
        xmid = zero + dx;
        fmid = f(xmid); // Evaluate function at the midpoint
        if (fmid <= 0.0) zero = xmid; // Lower bound refined
        if (fabs(dx) < TOL || fmid == 0.0) return zero; // Precision attained
    }

    cerr << "zerobisec : too many iterations" << endl;
    return 0;
}

// Returns a*(ab)/abs(ab) :
```

```
static double sgne(const double &a, const double &b)
{
    return (a * b >= 0) ? a : -a;
}

// Shifts a and b :
static void perm2(double &a, double &b)
{
    double tmp = a;
    a = b;
    b = tmp;
}

// Puts b into a, c into b, and d into c :
static void chang3(double &a, double &b, double &c, const double d)
{
    a = b;
    b = c;
    c = d;
}

// Brackets a minimum of function f :
static void minenc(double &ax, double &bx, double &cx, double f(const double))
{
    // Given initial bracketing, magnifies the interval so that actual bracketing
    const double PHI = 1.618034; // Default magnifying constant
    const double RLIMIT = 200.0; // Limit of parabolic interpolation
    const double EPS = 1.0e-20; // Precision
    double ulim, u, r, q, fa, fb, fc, fu;

    // Searches minimum in downhill direction defined by ax and bx.
    // Stops when starting going back uphill.
    fa = f(ax);
    fb = f(bx);
    if (fb > fa)
    {
        perm2(ax, bx);
        perm2(fb, fa); // Downhill direction defined to be from a to b.
    }
    cx = bx + PHI * (bx - ax); // Magnifying interval : going further downhill
    fc = f(cx);
```

```

while (fb > fc) // Third point not high enough : still going downhill
{
    // Tries parabolic interpolation
    r = (bx - ax) * (fb - fc);
    q = (bx - cx) * (fb - fa);
    // Optimum of the interpolated parabol located at u :
    u = bx - ((bx - cx) * q - (bx - ax) * r) / (2 * sgne(max(fabs(q - r), EPS)
    // Limit parabolic interpolation
    ulim = bx + RLIMIT * (cx - bx);

    if ((bx - u) * (u - cx) > 0) // u is between bx and cx
    {
        fu = f(u);
        if (fu < fc) // Minimum between bx and cx
        {
            ax = bx;
            bx = u; // Bracketing triplet is (bx,u,cx)
            return;
        }
        else if (fu > fb) // Minimum between ax and u
        {
            cx = u; // Bracketing triplet is (ax,bx,u)
            return;
        }
        u = cx + PHI * (cx - bx); // Parabolic interpolation was useless
        fu = f(u);
    }

    else if ((cx - u) * (u - ulim) > 0) // u is between cx and ulimit
    {
        fu = f(u);
        if (fu < fc)
        {
            chang3(bx, cx, u, u + PHI * (u - cx)); // Further downhill AND def
            chang3(fb, fc, fu, f(u));
        }
    }

    else if ((u - ulim) * (ulim - cx) >= 0) // Limits u to its maximum value
    {

```

```

        u = ulim;
        fu = f(u);
    }

    else
    {
        u = cx + PHI * (cx - bx); // Default magnification
        fu = f(u);
    }

    chang3(ax, bx, cx, u); // Continues further on downhill
    chang3(fa, fb, fc, fu);
}
}

```

```

// Finds a minimum of one-dimensional function f, bracketed by ax, bx, cx, with
static double min1dim(const double ax, const double bx, const double cx, double
{
    const int ITMAX = 1000; // Maximum nuber of iterations allowed
    const double PHI = 0.3819660; // Golden ratio : default step
    const double EPS = numeric_limits<double>::epsilon(); // Machine precision

    double a, b, d = 0.0, etemp, fu, fv, fw, fx;
    double p, q, r, tol1, tol2, u, v, w, x, xm;
    // x : where minimum value was found so far
    // w : where second least value was found
    // v : previous value of w
    // u : new trial point
    double e = 0.0;

    a = (ax < cx) ? ax : cx;
    b = (ax > cx) ? ax : cx; // Making a < b
    x = w = v = bx;
    fx = fw = fv = f(x);

    for (int i = 0; i < ITMAX; i++)
    {
        xm = 0.5 * (a + b); // [a,b] is the bracketing interval (refined at each i
        tol2 = 2.0 * (tol1 = tol * fabs(x) + EPS);
        if (fabs(x - xm) <= (tol2 - 0.5 * (b - a))) // Done : tolerance attained
        {

```

```

    xmin = x;
    return fx;
}

if (fabs(e) > tol1) // Parabolic interpolation using x,w,v
{
    r = (x - w) * (fx - fv);
    q = (x - v) * (fx - fw);
    p = (x - v) * q - (x - w) * r;
    q = 2.0 * (q - r);
    if (q > 0) p = -p;
    q = fabs(q);
    etemp = e;
    e = d;

    if (fabs(p) >= fabs(0.5 * q * etemp) || p <= q * (a - x) || p >= q * (
        // Parabolic interpolation rejected : default step
        d = PHI * (e = ((x > xm) ? a - x : b - x));
    else
    {
        d = p / q; // Parabolic step
        u = x + d;
        if (u - a < tol2 || b - u < tol2)
            d = sgne(tol1, xm - x);
    }

}

else d = PHI * (e = ((x >= xm) ? a - x : b - x)); // Default step

u = (fabs(d) >= tol1) ? x + d : x + sgne(tol1, d);
fu = f(u); // Only function evaluation in the loop

// Redefining bracketing triplet in each case
if (fu <= fx)
{
    if (u >= x) a = x;
    else b = x;
    chang3(v, w, x, u);
    chang3(fv, fw, fx, fu);
}

```

```

    else
    {
        if (u < x) a = u;
        else b = u;

        if (fu <= fw || w == x)
        {
            v = w;
            w = u;
            fv = fw;
            fw = fu;
        }
        else if (fu <= fv || v == x || v == w)
        {
            v = u;
            fv = fu;
        }
    }

}

}

cerr << "Too many iterations in min1dim" << endl;
xmin = x;
return fx;
}

// Global variables used for communication between "virtual" one-dimensional fun
// derived from function f in min1dir and routine min1dir

static int _n;
static double (*func)(double *);
static double *_p;
static double *_dir;

// One-dimensional virtual function derived from function func
// (which happens to be equal to function f in min1dir)
// in direction _dir

static double f1dim(const double x)

```

```

{
    double *xt = new double[_n];
    for (int j = 0; j < _n; j++)
    {
        xt[j] = _p[j] + x * _dir[j];
    }
    double val = func(xt);

    delete[] xt;
    return val;
}

// Finds a minimum of a multidimensional function f in direction dir.
// Minimum is stored in min, its location in p :
static void min1dir(int dim, double *p, double *dir, double &min, double f(double*)
{
    const double TOL = 1.0e-10;
    double xx, xmin, bx, ax;

    // Initialise les variables globales
    _n = dim;
    _p = new double[dim];
    _dir = new double[dim];
    func = f;

    for (int j = 0; j < dim; j++)
    {
        _p[j] = p[j];
        _dir[j] = dir[j];
    }

    // [0,1] is the initial bracketing guess
    ax = 0.0;
    xx = 1.0;
    minenc(ax, xx, bx, f1dim); // Computes an acutal bracketing triplet
    min = min1dim(ax, xx, bx, f1dim, TOL, xmin); // Computes the minimum of functi
    // (which is the minimum of function f in direction dir)

    for (int j = 0; j < dim; j++)
    {
        dir[j] *= xmin;
    }
}

```



```

    p[j] += dir[j]; // Sets actual position of the minimum
}

delete[] _dir;
delete[] _p;
}

// Conjugate gradient optimization of function f, given its gradient gradf.
// Minimum is stored in min, its location in p. Tolerance is asked :
static void optigc(int dim, double *p, const double tol, double &min, double f(d
{
    const int ITMAX = 20000;
    const double EPS = 1.0e-18;
    double gg, gam, fp, dgg; // Scalars used to define directions

    double *g = new double[dim]; // Auxiliary direction : gradient at the minimum
    double *h = new double[dim]; // Conjugate direction along which to minimize
    double *grad = new double[dim]; // Gradient

    fp = f(p);
    gradf(p, grad);

    for (int j = 0; j < dim; j++)
    {
        g[j] = -grad[j];
        grad[j] = h[j] = g[j];
    }

    for (int i = 0; i < ITMAX; i++)
    {
        min1dir(dim, p, h, min, f); // Minimizing along direction h
        if (2.0 * fabs(min - fp) <= tol * (fabs(min) + fabs(fp) + EPS))
        {
            delete[] g;
            delete[] h;
            delete[] grad;
            // Done : tolerance reached
            return;
        }
        fp = min;
        gradf(p, grad); // Computes gradient at point p, location of minimum
    }
}

```

```

    dgg = gg = 0.0;

    for (int j = 0; j < dim; j++) // Computes coefficients applied to new dire
    {
        gg += g[j] * g[j]; // Denominator
        dgg += (grad[j] + g[j]) * grad[j]; // Numerator : Polak-Ribiere
    }

    if (gg == 0.0) // Gradient equals zero : done
    {
        delete[] g;
        delete[] h;
        delete[] grad;
        return;
    }
    gam = dgg / gg;

    for (int j = 0; j < dim; j++) // Defining directions for next iteration
    {
        g[j] = -grad[j];
        h[j] = g[j] + gam * h[j];
    }

}

cerr << "Too many iterations in optigc" << endl;
}

// Global variables used for communication between Cost and Gradcost (Cout) func
// and low(up)linearprice routine, in which the functions are used

static int Dim;
static double *Eps;
static double *X;
static double *Rac_C;
static double Echeance;
static double *Sigma;

// Auxiliary cost function to minimize used in lowlinearprice :
static double Cost(double *ksi)

```

```

{
    double p = 0, arg = 0;

    double normv = 0;
    for (int i = 0; i < Dim + 1; i++)
    {
        normv += ksi[i] * ksi[i];
    }
    normv = sqrt(normv);

    for (int i = 0; i < Dim + 1; i++)
    {
        double tmp = 0;
        for (int j = 0; j < Dim + 1; j++)
        {
            tmp += Rac_C[i * (Dim + 1) + j] * ksi[j];
        }
        arg = ksi[Dim + 1] + Sigma[i] * tmp * sqrt(Echeance) / normv;
        p += Eps[i] * X[i] * cdf_nor(arg);
    }

    return (-1.0 * p); // The function is to be maximized
}

// Auxiliary gradient of function Cost, used in routine lowlinearprice :
static void Gradcost(double *ksi, double *g)
{
    double normv = 0;
    for (int i = 0; i < Dim + 1; i++)
    {
        normv += ksi[i] * ksi[i];
    }
    normv = sqrt(normv);

    g[Dim + 1] = 0;

    for (int j = 0; j < Dim + 1; j++)
    {
        g[j] = 0;
        for (int i = 0; i < Dim + 1; i++)
        {

```

```

        double tmp = 0;
        for (int k = 0; k < Dim + 1; k++)
        {
            tmp += Rac_C[i * (Dim + 1) + k] * ksi[k];
        }
        double s = pnl_normal_density(ksi[Dim + 1] + Sigma[i] * tmp * sqrt(Echeance));
        s *= Eps[i] * X[i];
        if (j == Dim)
            g[Dim + 1] += s;
        s *= Sigma[i] * sqrt(Echeance) / normv;
        s *= Rac_C[i * (Dim + 1) + j] - ksi[j] * tmp / (normv * normv);
        g[j] += s;
    }
    g[j] = -g[j];
}

g[Dim + 1] = -g[Dim + 1];
}

// Computes the price and the deltas of a claim using the lower bound of the price
// that is paying a linear combination of assets :
static void lowlinearprice(int _dim, double *_eps, double *_x, double *_rac_C, double *_ksi)
{
    // Initializing global variables to parameters of the problem
    Dim = _dim;
    Echeance = _echeance;

    Eps = new double[Dim + 1];
    for (int i = 0; i < Dim + 1; i++)
    {
        Eps[i] = _eps[i];
    }

    X = new double[Dim + 1];
    for (int i = 0; i < Dim + 1; i++)
    {
        X[i] = _x[i];
    }

    Rac_C = new double[(Dim + 1) * (Dim + 1)];
    for (int i = 0; i < Dim + 1; i++)

```

```
{
    for (int j = 0; j < Dim + 1; j++)
    {
        Rac_C[i * (Dim + 1) + j] = _rac_C[i * (Dim + 1) + j];
    }
}

Sigma = new double[Dim + 1];
for (int i = 0; i < Dim + 1; i++)
{
    Sigma[i] = _sigma[i];
}

// Starting point for optimization : normalized vector
double *xopt = new double[Dim + 2];
for (int i = 0; i < Dim + 1; i++)
{
    xopt[i] = 1. / sqrt(Dim + 1.);
}
xopt[Dim + 1] = 0;

double tol = 1e-15;

optigc(Dim + 2, xopt, tol, prix, Cost, Gradcost);

delete[] X;

prix = -1.0 * prix; // Price is the maximum of function

double normv = 0;
for (int i = 0; i < Dim + 1; i++)
{
    normv += xopt[i] * xopt[i];
}
normv = sqrt(normv);

for (int i = 0; i < Dim; i++)
{
    double tmp = 0;
    for (int j = 0; j < Dim + 1; j++)
```

```

        {
            tmp += Rac_C[(i + 1) * (Dim + 1) + j] * xopt[j];
        }
        double arg = xopt[Dim + 1] + Sigma[i + 1] * tmp * sqrt(Echeance) / normv;
        deltas[i] = Eps[i + 1] * cdf_nor(arg); // Computing the deltas
    }
    delete[] Rac_C;
    delete[] Eps;
    delete[] Sigma;
    delete[] xopt;
}

// Auxiliary cost function, the root of which is to be found in uplinearprice routine
static double Cout(const double ksi)
{
    double p = 0;

    for (int i = 0; i < Dim + 1; i++)
    {
        double arg = ksi + Eps[i] * Sigma[i] * sqrt(Echeance);
        p += Eps[i] * X[i] * pnl_normal_density(arg);
    }

    return p;
}

// Computes the price and the deltas of a claim using the upper bound of the price
// that is paying a linear combination of assets :
static void uplinearprice(int _dim, double *_eps, double *_x, double *sigmas, double *_echeance)
{
    // Initializing global variables to parameters of the problem
    Dim = _dim;
    Echeance = *_echeance;

    Eps = new double[Dim + 1];
    for (int i = 0; i < Dim + 1; i++)
    {
        Eps[i] = *_eps[i];
    }

    X = new double[Dim + 1];

```

```

for (int i = 0; i < Dim + 1; i++)
{
    X[i] = _x[i];
}

Sigma = new double[Dim + 1];

double *d = new double[Dim + 1]; // Roots of each function Cout defined in the
double *p = new double[Dim + 1]; // Associated "k-prices" (to be minimized in k

for (int k = 0; k < Dim + 1; k++)
{
    // Redefining sigma, thus redefining function Cout
    for (int i = 0; i < Dim + 1; i++)
    {
        Sigma[i] = sigmas[i * (Dim + 1) + k];
    }
    d[k] = zerobisec(Cout); // Finding the corresponding zero
    p[k] = 0;
    for (int i = 0; i < Dim + 1; i++) // Defining the associated "k-price"
    {
        double arg = d[k] + Eps[i] * Sigma[i] * sqrt(Echeance);
        p[k] += Eps[i] * X[i] * cdf_nor(arg);
    }
}

prix = p[0]; // Starting zero
int kopt = 0;

for (int k = 1; k < Dim + 1; k++)
{
    if (p[k] < prix) // Actual price is the minimum of all the virtual "k-prices"
    {
        prix = p[k];
        kopt = k;
    }
}

delete[] p;

double dopt = d[kopt];

```

```

delete[] d;
delete[] X;
delete[] Sigma;

for (int i = 0; i < Dim; i++)
{
    double arg = dopt + Eps[i + 1] * sigmas[(i + 1) * (Dim + 1) + kopt] * sqrt
    deltas[i] = Eps[i + 1] * cdf_nor(arg); // Defining deltas
}

delete[] Eps;
}

// Returning the price and the deltas of a discret-time average Asian option us
void lower_asian(int put_or_call, int dim, double vol, double val_init, double t
{
    // Initializing parameters
    PnlMat C_wrap;
    double *x = new double [dim + 1];
    x[0] = strike * exp(-tx_int * echeance);
    for (int i = 1; i < dim + 1; i++)
    {
        x[i] = 1.0 / dim * val_init * exp((tx_int - div) * i * echeance / dim - tx
    }
    double *eps = new double [dim + 1];
    if (put_or_call == 0)
    {
        eps[0] = -1;
        for (int i = 1; i < dim + 1; i++)
        {
            eps[i] = 1;
        }
    }
    else
    {
        eps[0] = 1;
        for (int i = 1; i < dim + 1; i++)
        {

```



```

        eps[i] = -1;
    }
}

double *C = new double [dim * dim]; // Correlation matrix
for (int i = 0; i < dim; i++)
{
    for (int j = 0; j < dim; j++)
    {
        if (i <= j)
            C[i * dim + j] = sqrt((1.0 * (i + 1)) / (j + 1));
        else C[i * dim + j] = sqrt((1.0 * (j + 1)) / (i + 1));
    }
}

C_wrap = pnl_mat_wrap_array(C, dim, dim);
pnl_mat_chol(&C_wrap);

double *rac_C = new double[(dim + 1) * (dim + 1)];
for (int i = 0; i < dim + 1; i++)
{
    rac_C[i * (dim + 1)] = 0;
    rac_C[i] = 0;
}
for (int i = 1; i < dim + 1; i++)
{
    for (int j = 1; j <= i; j++)
    {
        rac_C[i * (dim + 1) + j] = C[(i - 1) * dim + j - 1];
    }
    for (int j = i + 1; j < dim + 1; j++)
    {
        rac_C[i * (dim + 1) + j] = 0;
    }
}

delete[] C; // Correlation was useful only for computation of a square root of

double *sigma = new double[dim + 1];

```

```

sigma[0] = 0;
for (int i = 1; i < dim + 1; i++)
{
    sigma[i] = vol * sqrt((1.0 * i) / dim);
}

double *deltas = new double[dim];

lowlinearprice(dim, eps, x, rac_C, sigma, echeance, prix, deltas); // General

delta = 0;
for (int i = 0; i < dim; i++)
{
    delta += deltas[i] * x[i + 1] / val_init; // Linearity of derivation
}

delete[] deltas;
delete[] eps;
delete[] x;
delete[] rac_C;
delete[] sigma;
}

// Returning the price and the deltas of a discret-time average Asian option us
void upper_asian(int put_or_call, int dim, double vol, double val_init, double t
{
    // Initializing parameters
    double *x = new double [dim + 1];
    x[0] = strike * exp(-tx_int * echeance);
    for (int i = 1; i < dim + 1; i++)
    {
        x[i] = 1.0 / dim * val_init * exp((tx_int - div) * i * echeance / dim - tx
    }
    double *eps = new double [dim + 1];
    if (put_or_call == 0)
    {
        eps[0] = -1;
        for (int i = 1; i < dim + 1; i++)
        {
            eps[i] = 1;

```

```

        }
    }
    else
    {
        eps[0] = 1;
        for (int i = 1; i < dim + 1; i++)
        {
            eps[i] = -1;
        }
    }

    double *cov = new double [(dim + 1) * (dim + 1)];
    for (int i = 0; i < dim + 1; i++)
    {
        cov[i * (dim + 1)] = 0;
        cov[i] = 0;
    }
    for (int i = 1; i < dim + 1; i++)
    {
        for (int j = 1; j < dim + 1; j++)
        {
            if (i <= j) cov[i * (dim + 1) + j] = vol * vol * i * 1.0 / dim;
            else cov[i * (dim + 1) + j] = vol * vol * j * 1.0 / dim;
        }
    }

    double *sigmas = new double [(dim + 1) * (dim + 1)];
    for (int i = 0; i < dim + 1; i++)
    {
        for (int k = 0; k < dim + 1; k++)
        {
            sigmas[i * (dim + 1) + k] = cov[i * (dim + 1) + i] - cov[i * (dim + 1) + k];
            sigmas[i * (dim + 1) + k] = sqrt(sigmas[i * (dim + 1) + k]);
        }
    }

    delete[] cov; // Covariance matrix was useful only for computing sigma

    double *deltas = new double[dim + 1];

    uplinearprice(dim, eps, x, sigmas, echeance, prix, deltas); // General formula

```

```

delete[] eps;
delete[] sigmas;

delta = 0;
for (int i = 0; i < dim; i++)
{
    delta += deltas[i] * x[i + 1] / val_init; // Linearity of derivation
}

delete[] deltas;
delete[] x;
}

static int CarmonaDurlemaan_FixedAsian(double pseudo_stock, double pseudo_strike)
{
    double Deltainf = 0;
    double Prixinf = 0;
    int put_or_call;

    if ((p->Compute) == &Call_OverSpot2)
        put_or_call = 0;
    else
        put_or_call = 1;

    lower_asian(put_or_call, steps, sigma, pseudo_stock, r, divid, pseudo_strike,

/*upper_asian(put_or_call,steps,sigma,pseudo_stock,r,divid,pseudo_strike,t,Pri

/*Price*/
*ptprice_inf = Prixinf;

/*Delta */
*ptdelta_inf = Deltainf;

    return OK;
}
#endif //PremiaCurrentVersion

extern "C" {
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els

```

```

static int CHK_OPT(AP_FixedAsian_CarmonaDurleumann)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(AP_FixedAsian_CarmonaDurleumann)(void *Opt, void *Mod, PricingMethod *
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else
int CALC(AP_FixedAsian_CarmonaDurleumann)(void *Opt, void *Mod, PricingMethod *
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;

    int return_value;
    double r, divid, time_spent, pseudo_spot, pseudo_strike;
    double t_0, T_0;

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

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

    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
    {
        time_spent = (ptMod->T.Val.V_DATE - (ptOpt->PathDep.Val.V_NUMFUNC_2)->Pa
        pseudo_spot = (1. - time_spent) * ptMod->S0.Val.V_PDOUBLE;
        pseudo_strike = (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE -

        if (pseudo_strike <= 0.)
        {
            Fprintf(TOSCREEN, "ANALYTIC FORMULA\ n\ n\ n");
            return_value = Analytic_KemnaVorst(pseudo_spot, pseudo_strike, time_

```

```

        }
    else
        return_value = CarmonaDurlemaan_FixedAsian(pseudo_spot, pseudo_strike,
    }

    return return_value;
}

static int CHK_OPT(AP_FixedAsian_CarmonaDurlemaan)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "AsianCallFixedEuro") == 0) || (strcmp(((
        return OK;
    return WRONG;
}

#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;
        Met->Par[0].Val.V_INT2 = 200;
    }

    return OK;
}

PricingMethod MET(AP_FixedAsian_CarmonaDurlemaan) =
{
    "AP_FixedAsian_CarmonaDurlemaan",
    {"Nb of Monitoring Dates", INT2, {100}, ALLOW}, {" ", PREMIA_NULLTYPE, {0},
    CALC(AP_FixedAsian_CarmonaDurlemaan),
    {"Lower Price", DOUBLE, {100}, FORBID}, {"Lower Delta", DOUBLE, {100}, FORB
    CHK_OPT(AP_FixedAsian_CarmonaDurlemaan),
    CHK_ok,
    MET(Init)
};

```

}