

[Help](#)

```
#include <stdlib.h>
#include "
href../../mod/bs1d/bs1d_doublim/bs1d_doublim_h_src.pdfbs1d_doublim.h"
#include "
href../../common/error_msg_h_src.pdferror_msg.h"

static int Cryer_Out(double s, NumFunc_1 *L, NumFunc_1 *U, NumFunc_1 *Rebate,
{
    int      Index, PriceIndex, TimeIndex, ssl;
    double   k, vv, h, z, alpha, beta, gamma, y, down, u, l, rebate, up, upwind_al
    double   *Obst, *A, *B, *C, *P, *S, *Q, *Z, pricenh, pricen2h, priceph;

    /*Memory Allocation*/
    Obst = malloc((N + 1) * sizeof(double));
    if (Obst == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    A = malloc((N + 1) * sizeof(double));
    if (A == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    B = malloc((N + 1) * sizeof(double));
    if (B == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    C = malloc((N + 1) * sizeof(double));
    if (C == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    P = malloc((N + 1) * sizeof(double));
    if (P == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    S = malloc((N + 1) * sizeof(double));
    if (S == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    Z = malloc((N + 1) * sizeof(double));
    if (Z == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    Q = malloc((N + 1) * sizeof(double));
    if (Q == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    /*Time Step*/
```

```

k = t / (double)M;

/*Space Step*/
u = (U->Compute)(U->Par, 0);
l = (L->Compute)(L->Par, 0);
rebate = (Rebate->Compute)(Rebate->Par, 0);
y = log(s);
down = log(l);
up = log(u);
h = (up - down) / (double)(N);

/*Peclet Condition-Coefficient of diffusion augmented */
vv = 0.5 * SQR(sigma);
z = (r - divid) - vv;
if ((h * fabs(z)) <= vv)
    upwind_alphacoef = 0.5;
else
{
    if (z > 0.) upwind_alphacoef = 0.0;
    else upwind_alphacoef = 1.0;
}
vv -= z * h * (upwind_alphacoef - 0.5);

/*Lhs Factor of theta-schema*/
alpha = k * (-vv / (h * h) + z / (2.0 * h));
beta = 1.0 + k * (r + 2.*vv / (h * h));
gamma = k * (-vv / (h * h) - z / (2.0 * h));

for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
{
    A[PriceIndex] = alpha;
    B[PriceIndex] = beta;
    C[PriceIndex] = gamma;
}

/*Terminal Values*/
y = log(s);
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
    Obst[PriceIndex - 1] = (p->Compute)(p->Par, exp(down + PriceIndex * h));

for (PriceIndex = 2; PriceIndex <= N - 2; PriceIndex++)

```

```

{
    P[PriceIndex - 1] = alpha * Obst[PriceIndex - 2] +
                        beta * Obst[PriceIndex - 1] + gamma * Obst[PriceIndex]
}

P[0] = beta * Obst[0] + gamma * Obst[1];
P[N - 2] = alpha * Obst[N - 3] + beta * Obst[N - 2];

for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
{
    S[PriceIndex] = 0.0;
    Z[PriceIndex] = 0.0;
}
ssl = FALSE;

/*Finite Difference Cycle*/
for (TimeIndex = 1; TimeIndex <= M; TimeIndex++)
{
    for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
        Z[PriceIndex] = Z[PriceIndex] + Obst[PriceIndex];

    for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
        Q[PriceIndex] = P[PriceIndex] - Z[PriceIndex];
    Q[0] += alpha * rebate;
    Q[N - 2] += gamma * rebate;

    AlgCrayner(N, Z, ssl, A, B, C, Q, S);

    for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
        S[PriceIndex] = Z[PriceIndex];

    ssl = TRUE;
}

for (PriceIndex = 0; PriceIndex <= N - 2; PriceIndex++)
    P[PriceIndex] = Z[PriceIndex] + Obst[PriceIndex];

Index = (int)floor((y - down) / h) - 1;

/*Price*/
if ((y == up) && (y == down))

```

```

    *ptprice = P[0];
else
    *ptprice = P[Index] + (P[Index + 1] - P[Index]) * (exp(y) - exp(down + h + I

/*Delta*/
if ((y == up) && (y == down))
    *ptdelta = 0.0;
else
{
    pricen1h = P[Index + 1] + (P[Index + 2] - P[Index + 1]) * (exp(y + h) - exp
    if (Index > 0)
    {
        priceph = P[Index - 1] + (P[Index] - P[Index - 1]) * (exp(y - h) - exp
        *ptdelta = (pricen1h - priceph) / (2 * s * h);
    }
    else
    {
        pricen2h = P[Index + 2] + (P[Index + 3] - P[Index + 2]) * (exp(y + 2 *
        *ptdelta = (4 * pricen1h - pricen2h - 3 * (*ptprice)) / (2 * s * h);
    }
}

/*Memory Desallocation*/
free(Obst);
free(A);
free(B);
free(C);
free(P);
free(S);
free(Z);
free(Q);

return OK;
}

int CALC(FD_Cryer_Out)(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 Cryer_Out(ptMod->S0.Val.V_PDOUBLE,
                 ptOpt->LowerLimit.Val.V_NUMFUNC_1, ptOpt->UpperLimit.Val.V_NUMFUNC_1,
                 ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE, r, divid, p,
                 Met->Par[0].Val.V_INT, Met->Par[1].Val.V_INT,
                 &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE));
}

static int CHK_OPT(FD_Cryer_Out)(void *Opt, void *Mod)
{
    Option *ptOpt = (Option *)Opt;
    TYPEOPT *opt = (TYPEOPT *) (ptOpt->TypeOpt);

    if ((opt->Parisian).Val.V_BOOL == FALSE)
        if ((strcmp(((Option *)Opt)->Name, "DoubleCallOutAmer") == 0) || (strcmp(((Option *)Opt)->Name, "DoubleCallOutAmer") == 0))
            return OK;
    return WRONG;

    return WRONG;
}

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    if (Met->init == 0)
    {
        Met->init = 1;

        Met->Par[0].Val.V_INT2 = 100;
        Met->Par[1].Val.V_INT2 = 100;

    }

    return OK;
}

PricingMethod MET(FD_Cryer_Out) =
{
    "FD_Cryer_Out",

```

```

{"SpaceStepNumber", INT2, {100}, ALLOW}, {"TimeStepNumber", INT2, {100}, ALLOW},
CALC(FD_Cryer_Out),
{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID} , {" ", PR
CHK_OPT(FD_Cryer_Out),
CHK_split,
MET(Init)
};

```