

[Help](#)

```
#include "bs1d_lim.h"

static int Cryer_UpOut(double s, NumFunc_1 *p,
    double l, double rebate, double t, double r, double divid,
    double sigma, int N, int M, double *ptprice, double *ptde
    lta)
{
    int      Index, PriceIndex, TimeIndex, ssl, dummy;
    double   k, vv, loc, h, z, alpha, beta, gamma, y, up, upw
        ind_alphacoef;
    double   *Obst, *A, *B, *C, *P, *S, *Z, *Q, pricenh, pr
        icep2h, priceph;

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

```

/*Time Step*/
k=t/(double)M;

/*Space Localisation*/
vv=0.5*sigma*sigma;
z=(r-divid)-vv;
loc=sigma*sqrt(t)*sqrt(log(1.0/PRECISION))+fa
    bs(z*t);

/*Space Step*/
y=log(s);
up=log(1);
h=(up-(y-loc))/(double)(N);

/*Peclet Condition-Coefficient of diffusion augmented */
if ((h*fabs(z))<=vv)
    upwind_alphacoef=0.5;
else {
    if (z>0.) upwind_alphacoef=0.0;
    else if (z<=0.) 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(y-loc+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*(p->Compute)(p->Par,exp(y-loc));
    Q[N-2] += gamma*rebate;

    dummy=AlgCrayer(N,Z,ssl,A,B,C,Q,S);

    for (PriceIndex = 0; PriceIndex <=N-2; PriceIndex++)

```

```

    eIndex++)
        S[PriceIndex] = Z[PriceIndex];

    ssl = true;
}

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

Index = (int) floor(loc/h) - 1;

*ptprice = P[Index] + (P[Index+1] - P[Index]) * (exp(y) - exp(y - loc + h + Index*h)) / (exp(y - loc + h + (Index+1)*h) - exp(y - loc + h + Index*h));

/*Delta*/
priceph = P[Index-1] + (P[Index] - P[Index-1]) * (exp(y-h) - exp(y - loc + h + (Index-1)*h)) / (exp(y - loc + h + Index*h) - exp(y - loc + h + (Index-1)*h));
if (y != up) {
    pricen = P[Index+1] + (P[Index+2] - P[Index+1]) * (exp(y+h) - exp(y - loc + h + (Index+1)*h)) / (exp(y - loc + h + (Index+2)*h) - exp(y - loc + h + (Index+1)*h));
    *ptdelta = (pricen - priceph) / (2*s*h);
} else {
    pricep2h = P[Index-2] + (P[Index-3] - P[Index-2]) * (exp(y-2*h) - exp(y - loc + h + (Index-2)*h)) / (exp(y - loc + h + (Index-3)*h) - exp(y - loc + h + (Index-2)*h));
    *ptdelta = (-4*priceph + pricep2h + 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_UpOut)(void *Opt,void *Mod,PricingMethod *Met)
{
    TYPEOPT* ptOpt=(TYPEOPT*)Opt;
    TYPEMOD* ptMod=(TYPEMOD*)Mod;
    double r,divid,limit,rebate;

    r=log(1.+ptMod->R.Val.V_DOUBLE/100.);
    divid=log(1.+ptMod->Divid.Val.V_DOUBLE/100.);
    limit=((ptOpt->Limit.Val.V_NUMFUNC_1)->Compute)((ptOpt->Limit.Val.V_NUMFUNC_1)->Par,ptMod->T.Val.V_DATE);
    rebate=((ptOpt->Rebate.Val.V_NUMFUNC_1)->Compute)((ptOpt->Rebate.Val.V_NUMFUNC_1)->Par,ptMod->T.Val.V_DATE);

    return Cryer_UpOut(ptMod->S0.Val.V_PDOUBLE,ptOpt->PayOff.Val.V_NUMFUNC_1,
        limit,rebate, ptOpt->Maturity.Val.V_DATE-ptMod->T.Val.V_DATE,r,divid,ptMod->Sigma.Val.V_PDOUBBLE,
        Met->Par[0].Val.V_INT2,Met->Par[1].Val.V_INT2,
        &(Met->Res[0].Val.V_DOUBLE),&(Met->Res[1].Val.V_DOUBLE));
}

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

    if ((opt->Parisian).Val.V_BOOL==WRONG)

```

```

    if ( (strcmp( ((Option*)Opt)->Name,"
        CallUpOutAmer")==0) || (strcmp( ((Option*)Opt)->Name,"
        PutUpOutAmer")==0) )
        return OK;

    return WRONG;
}

static int MET(Init)(PricingMethod *Met)
{
    static int first=1;

    if (first)
    {
        Met->Par[0].Val.V_INT2=100;
        Met->Par[1].Val.V_INT2=100;

        first=0;
    }

    return OK;
}

PricingMethod MET(FD_Cryer_UpOut)=
{
    "FD_Cryer_UpOut",
    {{"SpaceStepNumber",INT2,100,ALLOW    },{"TimeStepNumber",INT2,100,ALLOW},
    {" ",END,0,FORBID}}},
    CALC(FD_Cryer_UpOut),
    {{"Price",DOUBLE,100,FORBID},{ "Delta",DOUBLE,100,FORBID} ,{" ",END,0,FORBID}}},
    CHK_OPT(FD_Cryer_UpOut),
    CHK_split,
    MET(Init)
};

```

References