

[Help](#)

```

#include <stdlib.h>
#include "bs1d_std.h"
#include "error_msg.h"
#define PRECISION 1.0e-7 /*Precision for the localization of FD methods*/

static int restriction1(int l, double *d, double *u, double *GG, double *A, double *B)
{
    int nl1, nl, i;
    double *aux;

    nl = pow(2, l + 1) - 1;

    aux = malloc((nl + 2) * sizeof(double));
    if (aux == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    nl1 = pow(2, l) - 1;

    for (i = 1; i < nl + 1; i++)
        aux[i] = u[i - 1] * A[i * N / (nl + 1)] + u[i] * B[i * N / (nl + 1)] + u[i + 1] * A[i * N / (nl + 1)];

    for (i = 1; i < nl1 + 1; i++)
        d[i] = aux[2 * i] / 2.0 + (aux[2 * i - 1] + aux[2 * i + 1]) / 4.0;

    free(aux);
    return OK;
}

static void substract_prolongation1(int l, double *u, double *v)
{
    int nl1, i;

    nl1 = pow(2, l) - 1;

    for (i = 0; i < nl1 + 1; i++)
    {

```

```

        u[2 * i] = u[2 * i] - v[i];
        u[2 * i + 1] = u[2 * i + 1] - (v[i] + v[i + 1]) / 2.0;
    }
    return;
}

```

```

static int MGM1(int l, double *u, double *GG, double *a, double *b, double *c, i
{
    int nl, nl1, i, j;
    double *v, *d;

    nl = pow(2, l + 1) - 1;
    nl1 = pow(2, l) - 1;

    v = malloc((nl1 + 2) * sizeof(double));
    if (v == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    d = malloc((nl1 + 2) * sizeof(double));
    if (d == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    if (l == 0) u[1] = GG[1] / b[N / 2];
    else
    {
        /* 2 iterations of Gauss-Seidel*/
        for (i = 1; i < 3; i++)
        {
            for (j = 1; j < nl + 1; j++)
                u[j] = (-u[j - 1] * a[j * N / (nl + 1)] - u[j + 1] * c[j * N / (nl +
            }

        restriction1(l, d, u, GG, a, b, c, N);

        for (i = 0; i < nl1 + 2; i++)
            v[i] = 0.0;

        MGM1(l - 1, v, d, a, b, c, N);

        subtract_prolongation1(l, u, v);
    }
}

```

```

        /* 2 iterations of Gauss-Seidel*/
        for (i = 1; i < 3; i++)
        {
            for (j = 1; j < nl + 1; j++)
                u[j] = (-u[j - 1] * a[j * N / (nl + 1)] - u[j + 1] * c[j * N / (nl + 1)]) / b[j * N / (nl + 1)];
        }

    free(v);
    free(d);
    return OK;
}

```

```

static int mult_amer1(double s, NumFunc_1 *p, double t, double r, double divid,
{
    double k, z, limit, h, x, alpha, beta, gamma, alpha1, beta1, gamma1, error, g0;
    double *P, *Obst, *R, *A, *B, *C, *GG, *u;
    int *pp;
    int i, j, Index, N;

    /*Memory Allocation*/
    N = pow(2, l + 1) - 1 + 1;
    P = malloc((N + 1) * sizeof(double));
    if (P == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    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;
    R = malloc((N + 1) * sizeof(double));

```

```

if (R == NULL)
    return MEMORY_ALLOCATION_FAILURE;
P = malloc((N + 1) * sizeof(double));
if (P == NULL)
    return MEMORY_ALLOCATION_FAILURE;
pp = malloc((N + 1) * sizeof(int));
if (pp == NULL)
    return MEMORY_ALLOCATION_FAILURE;
GG = malloc((N + 1) * sizeof(double));
if (GG == NULL)
    return MEMORY_ALLOCATION_FAILURE;
u = malloc((N + 1) * sizeof(double));
if (u == NULL)
    return MEMORY_ALLOCATION_FAILURE;

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

/*Space Localisation*/
z = (r - divid) - SQR(sigma) / 2.0;
limit = (sigma * sqrt(t) * sqrt(log(1.0 / PRECISION)) + fabs(z) * t);

/*Space Step*/
h = 2 * limit / (double)N;

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

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

```

```

alpha1 = k * (1.0 - theta) * (vv / (h * h) - z / (2.0 * h));
beta1 = 1.0 - k * (1.0 - theta) * (r + 2.*vv / (h * h));
gamma1 = k * (1.0 - theta) * (vv / (h * h) + z / (2.0 * h));

/*Terminal Values*/
x = log(s);
for (i = 0; i < N + 1; i++)
{
    Obst[i] = (p->Compute)(p->Par, exp(x - limit + i * h));
    P[i] = Obst[i];
}

/*Finite Difference Cycle*/
for (i = 1; i < M + 1; i++)
{
    /*Init pp and R*/
    for (j = 1; j < N; j++)
    {
        pp[j] = 0;
        R[j] = P[j] * beta1 + alpha1 * P[j - 1] + gamma1 * P[j + 1];
    }

    /*Howard Cycle*/
    do
    {
        error = 0.;

        for (j = 1; j < N; j++)
        {
            g0 = P[j - 1] * alpha + P[j] * beta + P[j + 1] * gamma - R[j];
            g1 = P[j] - Obst[j];
            if (g0 < g1) pp[j] = 0;
            else pp[j] = 1;
        }

        for (j = 1; j < N; j++)
        {
            if (pp[j] == 0)
            {
                GG[j] = R[j];
            }
        }
    } while (error > 0.);
}

```

```

        A[j] = alpha;
        B[j] = beta;
        C[j] = gamma;
    }
    else
    {
        GG[j] = Obst[j];
        A[j] = 0;
        B[j] = 1;
        C[j] = 0;
    }
}

for (j = 1; j < N; j++)
    u[j] = P[j];
MGM1(1, P, GG, A, B, C, N);
for (j = 1; j < N; j++)
    error += fabs(P[j] - u[j]);
}
while (error > epsilon);
/*End Howard Cycle*/
}
/*End Finite Difference Cycle*/

Index = (int) floor((double)N / 2.0);

/*Price*/
*ptprice = P[Index];

/*Delta*/
*ptdelta = (P[Index + 1] - P[Index - 1]) / (2.0 * s * h);

/*Memory Desallocation*/
free(P);
free(pp);
free(A);
free(B);
free(C);
free(R);
free(Obst);
free(GG);

```

```

    free(u);

    return OK;
}

```

```

int CALC(FD_FMGH)(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 mult_amer1(ptMod->S0.Val.V_PDOUBLE, ptOpt->PayOff.Val.V_NUMFUNC_1,
                      ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE, r, divid,
                      Met->Par[0].Val.V_INT, Met->Par[1].Val.V_INT, Met->Par[2].Val.V_INT,
                      Met->Par[3].Val.V_RGDOUBLE,
                      &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE));
}

```

```

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

    if ((opt->EuOrAm). Val.V_BOOL == AMER)
        return OK;

    return WRONG;
}

```

```

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

```

```

    Met->init = 1;

    Met->Par[0].Val.V_INT2 = 6;
    Met->Par[1].Val.V_INT2 = 128;
    Met->Par[2].Val.V_RGDOUBLE = 0.5;
    Met->Par[3].Val.V_RGDOUBLE = 0.000001;

}

return OK;
}

PricingMethod MET(FD_FMGH) =
{
    "FD_FMGH",
    { {"Number of Grids", INT2, {100}, ALLOW }, {"TimeStepNumber", INT2, {100},
        {"Theta", RGDOUBLE, {100}, ALLOW}, {"Epsilon", RGDOUBLE, {100}, ALLOW}, {" " "
    },
    CALC(FD_FMGH),
    {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID} , {" " ", PR
    CHK_OPT(FD_FMGH),
    CHK_fdiff,
    MET(Init)
};

```