

[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 Galerkin_Discontinuous(int am, double s, NumFunc_1 *p, double t, double
{
    double z, l, vv, b, c, a1, a2, a3, a4, error, y, norm, x_min, x_max, K;
    int i, j, Index1, Index2, loops, TimeIndex;
    double *P_Old, *Obst, *Rhs, *alpha4, *beta4, *gamma4, *alpha1, *beta1, *gamma1

    /*Memory Allocation*/
    alpha1 = malloc((2 * N + 2) * sizeof(double));
    if (alpha1 == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    beta1 = malloc((2 * N + 2) * sizeof(double));
    if (beta1 == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    gamma1 = malloc((2 * N + 2) * sizeof(double));
    if (gamma1 == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    alpha2 = malloc((2 * N + 2) * sizeof(double));
    if (alpha2 == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    beta2 = malloc((2 * N + 2) * sizeof(double));
    if (beta2 == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    gamma2 = malloc((2 * N + 2) * sizeof(double));
    if (gamma2 == NULL)
        return MEMORY_ALLOCATION_FAILURE;

    alpha3 = malloc((2 * N + 2) * sizeof(double));
    if (alpha3 == NULL)
        return MEMORY_ALLOCATION_FAILURE;

```

```
beta3 = malloc((2 * N + 2) * sizeof(double));
if (beta3 == NULL)
    return MEMORY_ALLOCATION_FAILURE;

gamma3 = malloc((2 * N + 2) * sizeof(double));
if (gamma3 == NULL)
    return MEMORY_ALLOCATION_FAILURE;

alpha4 = malloc((2 * N + 2) * sizeof(double));
if (alpha4 == NULL)
    return MEMORY_ALLOCATION_FAILURE;

beta4 = malloc((2 * N + 2) * sizeof(double));
if (beta4 == NULL)
    return MEMORY_ALLOCATION_FAILURE;

gamma4 = malloc((2 * N + 2) * sizeof(double));
if (gamma4 == NULL)
    return MEMORY_ALLOCATION_FAILURE;

alpha5 = malloc((2 * N + 2) * sizeof(double));
if (alpha5 == NULL)
    return MEMORY_ALLOCATION_FAILURE;

beta5 = malloc((2 * N + 2) * sizeof(double));
if (beta5 == NULL)
    return MEMORY_ALLOCATION_FAILURE;

gamma5 = malloc((2 * N + 2) * sizeof(double));
if (gamma5 == NULL)
    return MEMORY_ALLOCATION_FAILURE;

vec_tme = malloc((M + 1) * sizeof(double));
if (vec_tme == NULL)
    return MEMORY_ALLOCATION_FAILURE;

new_x = malloc((2 * N + 2) * sizeof(double));
if (new_x == NULL)
    return MEMORY_ALLOCATION_FAILURE;
```

```

P_Old = malloc((2 * N + 2) * sizeof(double));
if (P_Old == NULL)
    return MEMORY_ALLOCATION_FAILURE;

Obst = malloc((2 * N + 2) * sizeof(double));
if (Obst == NULL)
    return MEMORY_ALLOCATION_FAILURE;
Rhs = malloc((2 * N + 2) * sizeof(double));
if (Rhs == NULL)
    return MEMORY_ALLOCATION_FAILURE;

/*Space Localisation*/
y = log(s);
vv = 0.5 * SQR(sigma);
z = r - divid - vv;
l = sigma * sqrt(t) * sqrt(log(1.0 / PRECISION)) + fabs(z) * t;
K = p->Par[0].Val.V_PDOUBLE;

/*Terminal Values*/
x_min = y - l;
x_max = y + l;

for (i = 0; i <= M; i++)
    vec_tme[i] = ((double)i) * (t) / (double)M;

new_x[0] = x_min;
new_x[N] = x_max;
for (i = 0; i <= N; i++)
{
    new_x[i] = x_min + ((double)i) * (x_max - x_min) / (double)N;
    P_Old[i] = (p->Compute)(p->Par, exp(new_x[i]));
    Obst[i] = (p->Compute)(p->Par, exp(new_x[i]));
    P_Old[i + N] = 0.;
}
P_Old[2 * N + 1] = 0.;

/*Time Cycle*/
for (TimeIndex = 1; TimeIndex <= M; TimeIndex++)
{

```

```

a1 = (1. + r * (vec_tme[TimeIndex] - vec_tme[TimeIndex - 1])) / 6.;
b = vv * (vec_tme[TimeIndex] - vec_tme[TimeIndex - 1]);
c = z * (vec_tme[TimeIndex] - vec_tme[TimeIndex - 1]);

a2 = (1. + 0.5 * r * (vec_tme[TimeIndex] - vec_tme[TimeIndex - 1])) / 6.;
a3 = (0.5 * r * (vec_tme[TimeIndex] - vec_tme[TimeIndex - 1])) / 6.;
a4 = (1. / 2. + 1. / 3.*r * (vec_tme[TimeIndex] - vec_tme[TimeIndex - 1]))

/*Computation of Lhs coefficients*/
for (i = 1; i < N; i++)
{
    alpha1[i] = a1 * (new_x[i] - new_x[i - 1]) - b / (new_x[i] - new_x[i - 1])
    beta1[i] = 2.*a1 * ((new_x[i] - new_x[i - 1]) + (new_x[i + 1] - new_x[i]
        b * (1 / (new_x[i] - new_x[i - 1]) + 1. / (new_x[i + 1] - new_x[i]
    gamma1[i] = a1 * (new_x[i + 1] - new_x[i]) - b / (new_x[i + 1] - new_x[i]
}

for (i = 1; i < N; i++)
{
    alpha2[i] = a2 * (new_x[i] - new_x[i - 1]) - 0.5 * b / (new_x[i] - new_x[i - 1])
    beta2[i] = 2.*a2 * ((new_x[i] - new_x[i - 1]) + (new_x[i + 1] - new_x[i]
        b * (1 / (new_x[i] - new_x[i - 1]) + 1. / (new_x[i + 1] - new_x[i]
    gamma2[i] = a2 * (new_x[i + 1] - new_x[i]) - 0.5 * b / (new_x[i + 1] - new_x[i]
}

for (i = 1; i < N; i++)
{
    alpha3[i] = a3 * (new_x[i] - new_x[i - 1]) - 0.5 * b / (new_x[i] - new_x[i - 1])
    beta3[i] = 2.*a3 * ((new_x[i] - new_x[i - 1]) + (new_x[i + 1] - new_x[i]
        b * (1 / (new_x[i] - new_x[i - 1]) + 1. / (new_x[i + 1] - new_x[i]
    gamma3[i] = a3 * (new_x[i + 1] - new_x[i]) - 0.5 * b / (new_x[i + 1] - new_x[i]
}

for (i = 1; i < N; i++)
{
    alpha4[i] = a4 * (new_x[i] - new_x[i - 1]) - (1. / 3.) * b / (new_x[i] - new_x[i - 1])
    beta4[i] = 2.*a4 * ((new_x[i] - new_x[i - 1]) + (new_x[i + 1] - new_x[i]
        b * (1 / (new_x[i] - new_x[i - 1]) + 1. / (new_x[i + 1] - new_x[i]
    gamma4[i] = a4 * (new_x[i + 1] - new_x[i]) - (1. / 3.) * b / (new_x[i + 1] - new_x[i]
}

```

```

/*Computation of Rhs coefficients */
for (i = 1; i < N; i++)
{
    alpha5[i] = (new_x[i] - new_x[i - 1]) / 6.;
    beta5[i] = ((new_x[i] - new_x[i - 1]) + (new_x[i + 1] - new_x[i])) / 3;
    gamma5[i] = (new_x[i] - new_x[i - 1]) / 6.;
}

/*Init Rhs*/
for (j = 1; j < N; j++)
    Rhs[j] = alpha5[j] * (P_Old[j - 1] + P_Old[N + j]) + beta5[j] * (P_Old[j] + P_Old[N + j + 1]) + gamma5[j] * (P_Old[j - 1] - P_Old[N + j]);

if ((p->Compute) == &Call)
{
    P_Old[0] = 0.;
    P_Old[N] = exp(x_max) * exp(-divid * (vec_tme[TimeIndex - 1])) - K * exp(-divid * (vec_tme[TimeIndex - 1]));
    P_Old[N + 1] = 0.;
    P_Old[2 * N + 1] = exp(x_max) * exp(-divid * (vec_tme[TimeIndex])) - K * exp(-divid * (vec_tme[TimeIndex]));
}
else if ((p->Compute) == &Put)
{
    P_Old[0] = K * exp(-r * vec_tme[TimeIndex - 1]) - exp(x_min) * exp(-divid * (vec_tme[TimeIndex - 1]));
    P_Old[N] = 0.;
    P_Old[N + 1] = K * exp(-r * vec_tme[TimeIndex]) - exp(x_min) * exp(-divid * (vec_tme[TimeIndex]));
    P_Old[2 * N + 1] = 0.;
}

for (j = N + 2; j <= 2 * N; j++) Rhs[j] = 0.;

/*Psor Cycle*/
loops = 0;
do
{
    error = 0.;
    norm = 0.;

    for (j = 1; j < N; j++)
    {
        y = (Rhs[j] - alpha1[j] * P_Old[j - 1] - gamma1[j] * P_Old[j + 1]) / (1 - alpha1[j] - gamma1[j]);
        if (am)
            y = MAX(Obst[j], P_Old[j] + omega * (y - P_Old[j]));
        P_Old[j] = y;
    }
    error = MAX(error, norm);
    loops++;
} while (error > 1e-6);

```

```

        else
            y = P_Old[j] + omega * (y - P_Old[j]);

        error += (double)(j + 1) * fabs(y - P_Old[j]);

        norm += fabs(y);
        P_Old[j] = y;
    }

    for (j = N + 2; j <= 2 * N; j++)
    {
        y = (Rhs[j] - alpha4[j - N - 1] * P_Old[j - 1] - gamma4[j - N - 1]
        if (am)
            y = MAX(0., P_Old[j] + omega * (y - P_Old[j]));
        else
            y = P_Old[j] + omega * (y - P_Old[j]);

        error += (double)(j - 1) * fabs(y - P_Old[j]);
        norm += fabs(y);
        P_Old[j] = y;
    }

    if (norm < 1.0) norm = 1.0;
    error = error / norm;

    loops++;

}
while ((error > epsilon) && (loops < MAXLOOPS));
/*End Psor Cycle*/
}
/*End Finite Difference Cycle*/

Index1 = (int) floor((double)N / 2.0);
Index2 = (int) floor(N + 1 + (double)N / 2.0);

/*Price*/
*ptprice = P_Old[Index1] + P_Old[Index2];

/*Delta*/

```

```

*ptdelta = (P_Old[Index1 + 1] + P_Old[Index2 + 1] - (P_Old[Index1 - 1] + P_Old[Index2 - 1]) / 2);

/*Memory Desallocation*/
free(P_Old);
free(Obst);
free(Rhs);
free(alpha2);
free(beta2);
free(gamma2);
free(alpha1);
free(beta1);
free(gamma1);
free(alpha3);
free(beta3);
free(gamma3);
free(alpha4);
free(beta4);
free(gamma4);
free(alpha5);
free(beta5);
free(gamma5);
free(vec_tme);
free(new_x);

return OK;
}

int CALC(FD_Galerkin_Discontinuous)(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 Galerkin_Discontinuous(ptOpt->EuOrAm.Val.V_BOOL, ptMod->S0.Val.V_PDOUBLE,
                                   ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                                   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->Par[4].Val.V_RGDOUBLE,
                                   &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE));
}

```

```

}

static int CHK_OPT(FD_Galerkin_Discontinuous)(void *Opt, void *Mod)
{
    return OK;
}

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;
        Met->Par[2].Val.V_RGDOUBLE = 0.5;
        Met->Par[3].Val.V_RGDOUBLE = 1.5;
        Met->Par[4].Val.V_RGDOUBLE = 0.000001;

    }

    return OK;
}

PricingMethod MET(FD_Galerkin_Discontinuous) =
{
    "FD_Galerkin_DiscFem",
    { {"SpaceStepNumber", INT2, {100}, ALLOW }, {"TimeStepNumber", INT2, {100}, A
        {"Theta", RGDOUBLE051, {100}, ALLOW}, {"Omega", RGDOUBLE12, {100}, ALLOW},
    },
    CALC(FD_Galerkin_Discontinuous),
    { {"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID} , {" ", PR
    CHK_OPT(FD_Galerkin_Discontinuous),
    CHK_psor,
    MET(Init)
};

```