

[Help](#)

```

#include "bs2d_std2d.h"
#include "error_msg.h"
#define PRECISION 1.0e-7 /*Precision for the localization of FD methods*/

static int Psor(int am, double s1, double s2, NumFunc_2 *p, double t, double r,
{
    int TimeIndex, j, i, Index, n;
    double x1, x2, m1, m2, cov;
    double limit1, limit2, h1, h2;
    double a2, b2, c2, d2, e2, f2, g2, i2, j2;
    double k, y;
    double **P, **G, **Obst;
    double error, norm;
    int loops;

    /*Memory Allocation*/
    P = (double **)calloc(N + 1, sizeof(double *));
    if (P == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    for (i = 0; i < N + 1; i++)
    {
        P[i] = (double *)calloc(N + 1, sizeof(double));
        if (P[i] == NULL)
            return MEMORY_ALLOCATION_FAILURE;
    }
    G = (double **)calloc(N + 1, sizeof(double *));
    if (G == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    for (i = 0; i < N + 1; i++)
    {
        G[i] = (double *)calloc(N + 1, sizeof(double));
        if (G[i] == NULL)
            return MEMORY_ALLOCATION_FAILURE;
    }
    Obst = (double **)calloc(N + 1, sizeof(double *));
    if (Obst == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    for (i = 0; i < N + 1; i++)
    {

```

```

    Obst[i] = (double *)calloc(N + 1, sizeof(double));
    if (Obst[i] == NULL)
        return MEMORY_ALLOCATION_FAILURE;
}

m1 = (r - divid1) - SQR(sigma1) / 2.0;
m2 = (r - divid2) - SQR(sigma2) / 2.0;
cov = rho * sigma1 * sigma2;

/*Space Localisation*/
limit1 = sigma1 * sqrt(t) * sqrt(log(1 / PRECISION)) + fabs(m1) * t;
limit2 = sigma2 * sqrt(t) * sqrt(log(1 / PRECISION)) + fabs(m2) * t;

/*Space Step*/
h1 = 2.*limit1 / (double) N;
h2 = 2.*limit2 / (double)N;

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

/*Lhs factor*/
a2 = 1. + k * (r + SQR(sigma1) / SQR(h1) + SQR(sigma2) / SQR(h2));
b2 = -k * (SQR(sigma1) / (2.*SQR(h1)) + m1 / (2.*h1));
c2 = -k * (SQR(sigma1) / (2.*SQR(h1)) - m1 / (2.*h1));
d2 = -k * (SQR(sigma2) / (2.*SQR(h2)) + m2 / (2.*h2));
e2 = -k * (SQR(sigma2) / (2.*SQR(h2)) - m2 / (2.*h2));

f2 = k * cov / (4.*h1 * h2);
g2 = k * cov / (4.*h1 * h2);
i2 = -k * cov / (4.*h1 * h2);
j2 = -k * cov / (4.*h1 * h2);

/*Terminal Values*/
x1 = log(s1);
x2 = log(s2);

for (i = 0; i <= N; i++)
{
    for (j = 0; j <= N; j++)
    {
        Obst[i][j] = (p->Compute)(p->Par, exp(x1 - limit1 + h1 * (double)j),

```

```

                                exp(x2 + limit2 - h2 * (double)i));
        P[i][j] = Obst[i][j];
    }
}

/*Finite Difference Cycle */
for (TimeIndex = 1; TimeIndex <= M; TimeIndex++)
{
    for (i = 1; i < N; i++)
        for (n = 1; n < N; n++)
            G[i][n] = P[i][n];

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

        for (i = 1; i < N; i++)
        {
            for (n = 1; n < N; n++)
            {
                y = (G[i][n] - (c2 * P[i][n - 1] + b2 * P[i][n + 1] + e2 * P[i]
                    f2 * P[i + 1][n + 1] + g2 * P[i - 1][n - 1] +
                    j2 * P[i - 1][n + 1])) / a2;
                y = P[i][n] + omega * (y - P[i][n]);

                /*Projection for American case*/
                if (am)
                    y = MAX(Obst[i][n], y);

                error += fabs(y - P[i][n]);
                norm += fabs(y);
                P[i][n] = y;
            }
        }
        if (norm < 1.0) norm = 1.0;
        error = error / norm;
        loops++;
    }
}

```

```

        while ((error > epsilon) && (loops < MAXLOOPS));
        /*End Psor Cycle*/
        /* printf("%d\ n",loops);*/
    }

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

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

    /*Deltas*/
    *ptdelta1 = (P[Index][Index + 1] - P[Index][Index - 1]) / (2.*s1 * h1);
    *ptdelta2 = (P[Index - 1][Index] - P[Index + 1][Index]) / (2.*s2 * h2);

    /*Memory desallocation*/
    for (i = 0; i < N + 1; i++)
        free(P[i]);
    free(P);

    for (i = 0; i < N + 1; i++)
        free(G[i]);
    free(G);

    for (i = 0; i < N + 1; i++)
        free(Obst[i]);
    free(Obst);

    return OK;
}

int CALC(FD_Psor)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r, divid1, divid2;

    r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
    divid1 = log(1. + ptMod->Divid1.Val.V_DOUBLE / 100.);
    divid2 = log(1. + ptMod->Divid2.Val.V_DOUBLE / 100.);

    return Psor(ptOpt->EuOrAm.Val.V_BOOL, ptMod->S01.Val.V_PDOUBLE,

```

```

        ptMod->S02.Val.V_PDOUBLE, ptOpt->PayOff.Val.V_NUMFUNC_2,
        ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
        r, divid1, divid2, ptMod->Sigma1.Val.V_PDOUBLE, ptMod->Sigma2.Val.
        Met->Par[0].Val.V_INT, Met->Par[1].Val.V_INT, Met->Par[2].Val.V_RG
        &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].Val.V_DOUBLE), &(Met->R

    }

static int CHK_OPT(FD_Psor)(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_RGDOUBLE12 = 1.5;
        Met->Par[3].Val.V_RGDOUBLE = 0.000001;

    }
    return OK;
}

PricingMethod MET(FD_Psor) =
{
    "FD_Psor2d",
    { {"SpaceStepNumber", INT2, {100}, ALLOW}, {"TimeStepNumber", INT2, {100}, ALL
        , {"Omega", RGDOUBLE12, {100}, ALLOW}, {"Epsilon", RGDOUBLE, {100}, ALLOW},
    },
    CALC(FD_Psor),
    { {"Price", DOUBLE, {100}, FORBID}, {"Delta1", DOUBLE, {100}, FORBID} ,
        {"Delta2", DOUBLE, {100}, FORBID} ,
        {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(FD_Psor),
    CHK_ok,
    MET(Init)
}

```

};