

[Help](#)

```
extern "C" {
#include "
href../../mod/bs1d/bs1d_std/bs1d_std_h_src.pdfbs1d_std.h"
    extern char premia_data_dir[MAX_PATH_LEN];
    extern char *path_sep;
#include "pnl/pnl_vector.h"
#include "pnl/pnl_matrix.h"
}
#include <stdio.h>
#include <iostream>
#include <sstream>
#include <fstream>
#include <
href../../common/math/highdim_solver/highdim_vector_h_src.pdfvector>
#include <cmath>

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2010+2) //The "#els

extern "C" {
    static int CHK_OPT(FD_Trasparent)(void *Opt, void *Mod)
    {
        return NONACTIVE;
    }
    int CALC(FD_Trasparent)(void *Opt, void *Mod, PricingMethod *Met)
    {
        return AVAILABLE_IN_FULL_PREMIA;
    }
}
#else

using namespace std;

/*Class declarations*/

class TrasparentGrid
{
public :
    double xmin;
```

```

double xmax;
double dx;
int N;
double dt;
double T;
int M;

TrasparentGrid(const double dxmin, const double dxmax, const int dN, const double dx, const double dt, const double T, const int M)
{
    inline double x(double i) const
    {
        return xmin + i * dx;
    }
    inline double t(double n) const
    {
        return n * dt;
    }
};

/*Some useful routines*/

static double dot(const vector<double> &v1, const vector<double> &v2)
{
    const int n = v1.size();
    double result = 0;
    for (int i = 0; i < n; i++)
        result += v1[i] * v2[i];

    return (result);
}

/*=====*/
/* Brennan-Schwarz algorithms */
/*=====*/

vector<double> BrennanSchwartzPut_bckwd(vector<double> a, vector<double> b, vector<double> c, vector<double> d)
{
    /*realizes the backward step of the Brennan-Schwartz algorithm:
    for the system  $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$ , returns vector B such that
    where  $D_i = d_i - c_i D_{i+1} / B(i+1)$  but this is not included in the present routine
    Note: vectors a,b,c are of size N but elements a[0] and c[N-1] are not used*/

```

```

    int N = b.size();

    vector<double> B(N);
    B[N - 1] = b[N - 1];

    for (int i = N - 2; i >= 0; i--) B[i] = b[i] - a[i + 1] * c[i] / B[i + 1];

    return B;
}

vector<double> BrennanSchwartzPut_fwd(vector<double> c, vector<double> d, vector<double> a)
{
    /*computes vector D for the forward step of the Brennan-Schwartz algorithm for a put option
    for the system  $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$ , returns vector B such that  $B_i = d_i - a_i B_{i+1} / c_{i+1}$ 
    where  $D_i = d_i - a_i D_{i-1} / B(i-1)$  but this is not included in the present routine
    Note: vectors a,b,c are of size N but elements a[0] and c[N-1] are not used*/

    int N = d.size();

    vector<double> D(N);
    D[N - 1] = d[N - 1];

    for (int i = N - 2; i >= 0; i--) D[i] = d[i] - D[i + 1] * c[i] / B[i + 1];

    return D;
}

vector<double> BrennanSchwartzCall_fwd(vector<double> a, vector<double> b, vector<double> c)
{
    /*realizes the forward step of the Brennan-Schwartz algorithm for a call option
    for the system  $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$ , returns vector B such that  $B_i = d_i - a_i B_{i-1} / c_{i-1}$ 
    where  $D_i = d_i - a_i D_{i-1} / B(i-1)$  but this is not included in the present routine
    Note: vectors a,b,c are of size N but elements a[0] and c[N-1] are not used*/

    int N = b.size();

    vector<double> B(N);
    B[0] = b[0];

    for (int i = 1; i < N; i++) B[i] = b[i] - a[i] * c[i - 1] / B[i - 1];

    return B;
}

```

```

vector<double> BrennanSchwartzCall_bckwd(vector<double> a, vector<double> d, vec
{
    /*computes vector D for the backward step of the Brennan-Schwartz algorithm fo

    int N = d.size();

    vector<double> D(N);
    D[0] = d[0];

    for (int i = 1; i < N; i++) D[i] = d[i] - D[i - 1] * a[i] / B[i - 1];

    return D;
}
/*=====*/
/*LU solver for linear algebraic equations */
/*=====*/

static void lusolver(vector<vector<double> > &A, vector<double> &b)
{
    PnlVect Vb = pnl_vect_wrap_array(&(b[0]), b.size());
    int i, n = b.size();

    PnlMat *M;
    M = pnl_mat_create(A.size(), n);
    for (i = 0; i < n; i++)
        memcpy(&(M->array[i * n]), &(A[i][0]), n * sizeof(double));
    pnl_mat_syslin_inplace(M, &Vb);
    pnl_mat_free(&M);
}

/*=====*/
/*Pade approximation parameters for boundary conditions
Create file InterpolationParameters.
This file is in Premia/data directory */
/*=====*/

//void create_interp_params(const char * filename)
//{
//    /*creates a text file with name filename which contains interpolation paramet
//    F[i] = sqrt(z[i]), M[i][j] = z[i]/(z[i]+z[j]);

```

```

// here the interpolation points  $z[i]$  are  $1, 2, 1/2, 4, 1/4, 8, 1/8, \dots$  but this may b
// const int Nmax = 30;
//
// vector<double> z(Nmax);
// z[0] = 1;
// for (int i=1,k=1;i<Nmax;i=i+2,k++)
// {
//   z[i] = pow(2.,k);
//   if(i+1<Nmax)
//   {
//     z[i+1] = pow(2.,-k);
//   }
// }
//
// ofstream intparam(filename);
// for (int i=0;i<Nmax;i++)
// {
//   intparam << z[i] << " " << sqrt(z[i]) << endl;
//   for(int j=0;j<Nmax;j++)
//   {
//     intparam << z[i]/(z[i]+z[j]) << " ";
//   }
//   intparam << endl;
// }
//}

```

```

static void read_interp_params(const char *filename, const int ml, const int mr,
                               double &ksi0, vector<double> &alpha, vector<double>
                               double &eta0, vector<double> &beta, vector<double>
{
    ifstream intparams(filename);

    if (ml == mr)
    {
        int n = ml;
        vector<double> F(n), z(n);
        vector<vector<double> > M(n, vector<double>(n));
        for (int i = 0; i < n; i++)
        {

```

```

        intparams >> z[i];
        intparams >> F[i];
        for (int j = 0; j < n; j++)
        {
            intparams >> M[i][j];
        }
        intparams.ignore(1000000, '\ n');
    }

    lusolver(M, F);

    double sum = 0;
    const double sqrt2 = sqrt(2.);
    for (int i = 0; i < n; i++)
    {
        sum += F[i];
        alpha[i] = -sqrt2 / sigma * F[i] * z[i];
        beta[i] = -alpha[i];
        gamma[i] = mu * mu / (2 * sigma * sigma) + z[i];
        delta[i] = gamma[i];
    }
    ksi0 = -mu / (sigma * sigma) + sqrt2 / sigma * sum;
    eta0 = -mu / (sigma * sigma) - sqrt2 / sigma * sum;
}
else
{
    int n = m1;
    vector<double> F1(n), z1(n);
    vector<vector<double>> > M1(n, vector<double>(n));
    for (int i = 0; i < n; i++)
    {
        intparams >> z1[i];
        intparams >> F1[i];
        for (int j = 0; j < n; j++)
        {
            intparams >> M1[i][j];
        }
        intparams.ignore(1000000, '\ n');
    }
}

```

```

lusolver(Ml, Fl);

double sum = 0;
const double sqrt2 = sqrt(2.);
for (int i = 0; i < n; i++)
{
    sum += Fl[i];
    alpha[i] = -sqrt2 / sigma * Fl[i] * zl[i];
    gamma[i] = mu * mu / (2 * sigma * sigma) + zl[i];
}
ksi0 = -mu / (sigma * sigma) + sqrt2 / sigma * sum;

n = mr;
vector<double> Fr(n), zr(n);
vector<vector<double> > Mr(n, vector<double>(n));
intparams.close();

ifstream intparams(filename);
for (int i = 0; i < n; i++)
{
    intparams >> zr[i];
    intparams >> Fr[i];
    for (int j = 0; j < n; j++)
    {
        intparams >> Mr[i][j];
    }
    intparams.ignore(1000000, '\\ n');
}

lusolver(Mr, Fr);

sum = 0;
for (int i = 0; i < n; i++)
{
    sum += Fr[i];
    beta[i] = sqrt2 / sigma * Fr[i] * zr[i];
    delta[i] = mu * mu / (2 * sigma * sigma) + zr[i];
}
eta0 = -mu / (sigma * sigma) - sqrt2 / sigma * sum;
}
}

```

```

/*=====*/
/*Finite difference scheme with approximate transparent boundary conditions*/
/*=====*/
TrasparentGrid::TrasparentGrid(const double dxmin, const double dxmax, const int
:
    xmin(dxmin), xmax(dxmax), N(dN), T(dT), M(dM)
{
    dx = (xmax - xmin) / (N - 1);
    dt = T / M;
}

static void BS2thetaFD(const double theta, const double r, const double sigma, c
    double &al, double &ad, double &au, double &bl, double &bu
{

    double ss2 = sigma * sigma / 2;
    double mu = r - divid - ss2;

    double cl = -ss2 / (dx * dx) + mu / (2 * dx);
    double cd = ss2 * 2. / (dx * dx);
    double cu = -ss2 / (dx * dx) - mu / (2 * dx);

    al = dt * theta * cl;
    ad = 1 + dt * theta * cd;
    au = dt * theta * cu;

    bl = -dt * (1 - theta) * cl;
    bd = 1 - dt * (1 - theta) * cd;
    bu = -dt * (1 - theta) * cu;
}

static int Trasparent(int am, double S0, NumFunc_1 *p, double T, double r, doub
{
    double xmin, xmax;
    double q;

    xmin = log(Smin / S0);
    xmax = log(Smax / S0);
}

```



```

double K = p->Par[0].Val.V_PDDOUBLE;
if (((p->Compute) == &Put) && (am == 1))
{
    q = (0.5 - (r - divid) / SQR(sigma)) - sqrt((r - divid) / SQR(sigma) * (r
    xmin = log(K * q / (q - 1) / S0);
    cerr << "S_Min changed in the American Case" << endl;
}
else if (((p->Compute) == &Call) && (am == 1))
{
    q = (0.5 - (r - divid) / SQR(sigma)) + sqrt((r - divid) / SQR(sigma) * (r
    xmax = log(K * q / (q - 1) / S0);
    cerr << "S_Max changed in the American Case" << endl;
}

const TransparentGrid grid(xmin, xmax, Nspace, T, Ntime);

//coefficients of the finite difference scheme
double al, ad, au, bl, bd, bu;
BS2thetaFD(theta, r, sigma, divid, grid.dx, grid.dt, al, ad, au, bl, bd, bu);

//parameters of the Pade approximation of boundary conditions
double ksi0, eta0;
vector<double> alpha(ml), beta(mr), gamma(ml), delta(mr);

double mucoef = r - divid - sigma * sigma / 2;
std::string path(premia_data_dir);
path += path_sep;
std::ifstream intparams((path + "InterpolationParameters.txt").c_str());

if (!intparams)
    return UNABLE_TO_OPEN_FILE;

read_interp_params((path + "InterpolationParameters.txt").c_str(), ml, mr, mucoef);

//some auxiliary coefficients
vector<double> alphagamma(ml), betadelta(mr);
double suml = 0, sumr = 0;

for (int j = 0; j < ml; j++)
{
    alphagamma[j] = alpha[j] / (1 + grid.dt / 2 * gamma[j]);

```

```

        suml += alphagamma[j];
    }
    for (int j = 0; j < mr; j++)
    {
        betadelta[j] = beta[j] / (1 + grid.dt / 2 * delta[j]);
        sumr += betadelta[j];
    }

    //taking into account non-homogeneous boundary conditions
    //this will appear in the right-hand side
    vector<vector<double>> > Vmin(Ntime + 1, vector<double>(3)), Vmax(Ntime + 1, ve
    vector<vector<double>> > mu(Ntime + 1, vector<double>(ml)), omega(Ntime + 1, v

    if ((p->Compute) == &Call) //Call
    {
        for (int i = 0; i < 3; i++)
            for (int n = 0; n < Ntime + 1; n++) Vmax[n][i] = S0 * exp(grid.x(i + Nsp
        for (int j = 0; j < mr; j++) omega[0][j] = S0 * exp(xmax) / (r - divid + d
    }
    else
    {
        // Put
        for (int i = 0; i < 3; i++)
            for (int n = 0; n < Ntime + 1; n++) Vmin[n][i] = K - S0 * exp(grid.x(i -
        for (int j = 0; j < ml; j++) mu[0][j] = K / gamma[j] - S0 * exp(xmin) / (r
    }

    for (int n = 1; n < Ntime + 1; n++)
    {
        for (int j = 0; j < ml; j++) mu[n][j] = 1. / (1 + grid.dt / 2.*gamma[j]) *
        for (int j = 0; j < mr; j++) omega[n][j] = 1. / (1 + grid.dt / 2.*delta[j]
    }

    //construction of the tridiagonal matrix
    vector<double> ldiag(Nspace, al), diag(Nspace, ad), udiag(Nspace, au);

    const double cl = 2 * grid.dx * (ksi0 + grid.dt / 2 * suml);
    diag[0] = ad - cl * al;
    udiag[0] = au + al;

```

```

ldiag[Nspace - 1] = al + au;
const double cr = 2 * grid.dx * (eta0 + grid.dt / 2 * sumr);
diag[Nspace - 1] = ad + cr * au;

vector<double> u(Nspace), v(Nspace);
double ul, ur, umin, umax;
vector<double> lambda(ml), rho(mr);

/* initial conditions */
for (int i = 0; i < Nspace; i++) u[i] = (p->Compute)(p->Par, S0 * exp(grid.x(i)));
ul = (p->Compute)(p->Par, S0 * exp(grid.x(-1)));
ur = (p->Compute)(p->Par, S0 * exp(grid.x(Nspace)));

for (int j = 0; j < ml; j++) lambda[j] = mu[0][j];
for (int j = 0; j < mr; j++) rho[j] = omega[0][j];

vector<double> L(ml), R(mr);
double Dvmin, Dvmax;

if (am == 0) //European call or put
{
    vector<double> B = BrennanSchwartzPut_bckwd(ldiag, diag, udiag);
    for (int n = 0; n < Ntime; n++) //time iterations
    {
        /*computation of the right-hand side vector v */
        for (int j = 0; j < ml; j++)
        {
            L[j] = lambda[j] + grid.dt / 2 * (u[0] - gamma[j] * lambda[j]);
        }
        for (int j = 0; j < mr; j++)
        {
            R[j] = rho[j] + grid.dt / 2 * (u[Nspace - 1] - delta[j] * rho[j]);
        }
        Dvmin = (Vmin[n + 1][2] - Vmin[n + 1][0]) / (2 * grid.dx) - ksi0 * Vmi
        Dvmax = (Vmax[n + 1][2] - Vmax[n + 1][0]) / (2 * grid.dx) - eta0 * Vma

        v[0] = bl * ul + bd * u[0] + bu * u[1] + 2 * grid.dx * (dot(alphagamma
        for (int i = 1; i < Nspace - 1; i++)
        {
            v[i] = bl * u[i - 1] + bd * u[i] + bu * u[i + 1];

```

```

    }
    v[Nspace - 1] = bl * u[Nspace - 2] + bd * u[Nspace - 1] + bu * ur - 2 * u[Nspace - 1];

    /*saving u^n at xmin and xmax before computing u^{n+1}*/
    umin = u[0];
    umax = u[Nspace - 1];

    /*computation of u^{n+1}*/
    vector<double> D = BrennanSchwartzPut_fwd(udiag, v, B);

    u[0] = D[0] / B[0];

    for (int i = 1; i < Nspace; i++)
    {
        u[i] = (D[i] - ldiag[i] * u[i - 1]) / B[i];
    }

    /*updating the coefficients of the right-hand side*/
    for (int j = 0; j < ml; j++) lambda[j] = (lambda[j] + grid.dt / 2 * (u[Nspace - 1] - u[0]) * gamma[j]);
    ul = u[1] - 2 * grid.dx * (ksi0 * u[0] + dot(alpha, lambda) + Dvmin);

    for (int j = 0; j < mr; j++) rho[j] = (rho[j] + grid.dt / 2 * (u[Nspace - 1] - u[0]) * delta[j]);
    ur = u[Nspace - 2] + 2 * grid.dx * (eta0 * u[Nspace - 1] + dot(beta, rho));

} //end of time iterations
}
else if ((p->Compute) == &Put) //American put
{
    vector<double> B = BrennanSchwartzPut_bckwd(ldiag, diag, udiag);
    double payoff, exprt;
    for (int n = 0; n < Ntime; n++) //time iterations
    {
        /*computation of the right-hand side vector v */
        for (int j = 0; j < ml; j++)
        {
            L[j] = lambda[j] + grid.dt / 2 * (u[0] - gamma[j] * lambda[j]);
        }
        for (int j = 0; j < mr; j++)
        {
            R[j] = rho[j] + grid.dt / 2 * (u[Nspace - 1] - delta[j] * rho[j]);
        }
    }
}

```

```

Dvmin = (Vmin[n + 1][2] - Vmin[n + 1][0]) / (2 * grid.dx) - ksi0 * Vmi
Dvmax = (Vmax[n + 1][2] - Vmax[n + 1][0]) / (2 * grid.dx) - eta0 * Vma

v[0] = bl * ul + bd * u[0] + bu * u[1] + 2 * grid.dx * (dot(alphagamma
for (int i = 1; i < Nspace - 1; i++)
{
    v[i] = bl * u[i - 1] + bd * u[i] + bu * u[i + 1];
}
v[Nspace - 1] = bl * u[Nspace - 2] + bd * u[Nspace - 1] + bu * ur - 2

/*saving u^n at xmin and xmax before computing u^{n+1}*/
umin = u[0];
umax = u[Nspace - 1];

/*computation of u^{n+1}*/
vector<double> D = BrennanSchwartzPut_fwd(udiag, v, B);

exprrt = exp(r * grid.t(n + 1));
payoff = exprrt * (K - S0 * exp(grid.x(0)));
u[0] = MAX(D[0] / B[0], payoff);

for (int i = 1; i < Nspace; i++)
{
    payoff = exprrt * (K - S0 * exp(grid.x(i)));
    u[i] = MAX((D[i] - ldiag[i] * u[i - 1]) / B[i], payoff);
}

/*updating the coefficients of the right-hand side*/
for (int j = 0; j < ml; j++) lambda[j] = (lambda[j] + grid.dt / 2 * (u
ul = u[1] - 2 * grid.dx * (ksi0 * u[0] + dot(alpha, lambda) + Dvmin);

for (int j = 0; j < mr; j++) rho[j] = (rho[j] + grid.dt / 2 * (u[Nspace
ur = u[Nspace - 2] + 2 * grid.dx * (eta0 * u[Nspace - 1] + dot(beta, r

} //end of time iterations
}
else //American call
{
    vector<double> B = BrennanSchwartzCall_fwd(ldiag, diag, udiag);
    double payoff, exprrt;

```

```

for (int n = 0; n < Ntime; n++) //time iterations
{
    /*computation of the right-hand side vector v */
    for (int j = 0; j < ml; j++)
    {
        L[j] = lambda[j] + grid.dt / 2 * (u[0] - gamma[j] * lambda[j]);
    }
    for (int j = 0; j < mr; j++)
    {
        R[j] = rho[j] + grid.dt / 2 * (u[Nspace - 1] - delta[j] * rho[j]);
    }
    Dvmin = (Vmin[n + 1][2] - Vmin[n + 1][0]) / (2 * grid.dx) - ksi0 * Vmi
    Dvmax = (Vmax[n + 1][2] - Vmax[n + 1][0]) / (2 * grid.dx) - eta0 * Vma

    v[0] = bl * ul + bd * u[0] + bu * u[1] + 2 * grid.dx * (dot(alphagamma
    for (int i = 1; i < Nspace - 1; i++)
    {
        v[i] = bl * u[i - 1] + bd * u[i] + bu * u[i + 1];
    }
    v[Nspace - 1] = bl * u[Nspace - 2] + bd * u[Nspace - 1] + bu * ur - 2

    /*saving u^n at xmin and xmax before computing u^{n+1}*/
    umin = u[0];
    umax = u[Nspace - 1];

    /*computation of u^{n+1}*/
    vector<double> D = BrennanSchwartzCall_bckwd(ldiag, v, B);

    exprt = exp(r * grid.t(n + 1));
    payoff = exprt * (S0 * exp(grid.x(0)) - K);
    u[Nspace - 1] = MAX(D[Nspace - 1] / B[Nspace - 1], payoff);

    for (int i = Nspace - 2; i >= 0; i--)
    {
        payoff = exprt * (S0 * exp(grid.x(i)) - K);
        u[i] = MAX((D[i] - udiag[i] * u[i + 1]) / B[i], payoff);
    }

    /*updating the coefficients of the right-hand side*/
    for (int j = 0; j < ml; j++) lambda[j] = (lambda[j] + grid.dt / 2 * (u

```



```

        return Trasparent(ptOpt->EuOrAm.Val.V_BOOL, ptMod->S0.Val.V_PDOUBLE,
                           ptOpt->PayOff.Val.V_NUMFUNC_1, ptOpt->Maturity.Val.V_DATE,
                           Met->Par[0].Val.V_INT, Met->Par[1].Val.V_INT, Met->Par[2].Val.V_INT);
    }

static int CHK_OPT(FD_Trasparent)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt)->Name, "PutEuro") == 0))
        return OK;

    return WRONG;
}

#endif //PremiaCurrentVersion
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_DOUBLE = 80;
        Met->Par[4].Val.V_DOUBLE = 120;
        Met->Par[5].Val.V_RGINT130 = 5;
        Met->Par[6].Val.V_RGINT130 = 5;
    }

    return OK;
}

PricingMethod MET(FD_Trasparent) =
{
    "FD_Trasparent",
    { {"SpaceStepNumber", INT2, {100}, ALLOW }, {"TimeStepNumber", INT2, {100},
        {"Theta", RGDOUBLE051, {100}, ALLOW},
        {"S_Min", DOUBLE, {100}, ALLOW},
        {"S_Max", DOUBLE, {100}, ALLOW},
        {"Number of terms in Pade expansion at S_Min", RGINT130, {100}, ALLOW},
        {"Number of terms in Pade expansion at S_Max", RGINT130, {100}, ALLOW},
        {" ", PREMIA_NULLTYPE, {0}, FORBID}

```



```

    },
    CALC(FD_Trasparent),
    { {"Price", DOUBLE, {100}, FORBID},
      {"Delta", DOUBLE, {100}, FORBID} ,
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(FD_Trasparent),
    CHK_split,
    MET(Init)
};
}

```