Help

```
#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
#else

#include <iostream>
#include <
href../../common/math/highdim_solver/highdim_vector_h_src.pdfvector>
#include <cmath>

using namespace std;

#include "
href../../common/math/fft_h_src.pdffft.h"
#include "
href../../common/math/numerics_h_src.pdfnumerics.h"
#include "
href../../common/math/levy_fd_swing_h_src.pdflevy_fd_swing.h"

static double ** **Price;
static int mat_index;
static double *low, *diag, *up;
static double *low_obst, *diag_obst, *up_obst;
static double *Obst;
static double *Vp, *beta_p, *u, *v;
static int N_fft;
static int p;
static int NN_fft;
static int Nz ;
static double *mu;
static double *mu_img ;
static double *uaux;
static double *uaux_img;
static double *somme;
static double *somme_img;
static double a, b, c;

static int init_matrix(double dx, double dt, double ss, double aux, double r, in
{

  double dxx = dx * dx;
```

```
  if (dx < ss / fabs(aux))
    {
      a = ss / dxx + r;
      b = ss / 2. / dxx - aux / 2. / dx;
      c = ss / 2. / dxx + aux / 2. / dx;
    }
  else if (aux < 0)
    {
      a = ss / dxx - aux / dx + r;
      b = ss / 2. / dxx - aux / dx;
      c = ss / 2. / dxx;
    }
  else
    {
      a = ss / dxx + aux / dx + r;
      b = ss / 2. / dxx;
      c = ss / 2. / dxx + aux / dx;
    }

  for (int i = 0; i < Nspace; i++)
    {
      low[i] = -dt * c;
      diag[i] = 1 + dt * a;
      up[i] = -dt * b;
    }
  return OK;
}

static int init_matrix_obst(double dx, double dt, double ss, double aux, double
{
  double dxx = dx * dx;

  if (dx < ss / fabs(aux))
    {
      a = ss / dxx + r;
      b = ss / 2. / dxx - aux / 2. / dx;
      c = ss / 2. / dxx + aux / 2. / dx;
    }
  else if (aux < 0)
    {
```

```
      a = ss / dxx - aux / dx + r;
      b = ss / 2. / dxx - aux / dx;
      c = ss / 2. / dxx;
    }
  else
    {
      a = ss / dxx + aux / dx + r;
      b = ss / 2. / dxx;
      c = ss / 2. / dxx + aux / dx;
    }

  for (int i = 0; i < Nspace; i++)
    {
      low_obst[i] = -dt * c;
      diag_obst[i] = 1 + dt * a;
      up_obst[i] = -dt * b;
    }
  return OK;
}

GridSwing::GridSwing(const double dAl, const double dAr, const int dN)
  :
  Al(dAl), Ar(dAr), N(dN)
{
  dx = (Ar - Al) / (N - 1);
}

double init_cond_swing(const double x, const double S0,
                       const double K, const int product)
{
  double S = S0 * exp(x);

  switch (product)
    {
    case 1:
      return (S - K > 0) ? (S - K) : 0; // Call
    case 2:
      return (K - S > 0) ? (K - S) : 0; // Put
    case 3:
      return  S - K; // forward
    default:
```

```
      myerror("Invalid product number");
    }
  /* just to avoid a warning */
  return 0;
}

double bound_cond_swing(const double x, const double S0, const double K, const d
                        const double ttm, const double r,
                        const int product, const int product_type)
{
  switch (product_type)
    {
    case 1:
      return init_cond_swing(x + r * ttm, S0, K, product); // European
    case 2:
      return (x > 0) ? rebate * exp(r * ttm) : init_cond_swing(x + r * ttm, S0,
    case 3:
      return (x > 0) ? init_cond_swing(x + r * ttm, S0, K, product) : rebate * e
    case 4:
      return rebate * exp(r * ttm); // Double barrier
    default:
      myerror("Invalid option type number");
    }
  /* just to avoid a warning */
  return 0;
}

static int compute_obst(const Levy_measure &measure, int product,
                        const int product_type, double r, double divid,
                        double S0, double K, double rebate, double Al,
                        double Ar, int Nspace, double eval_date, int Ntime, int
{


  /*Maturita*/
  if ((nu == 0) && (nd == 0))
    {
      for (int i = 0; i < Nspace; i++)
        Obst[i] = 0.;
      return 1;
    }
```

```
double mat_date = eval_date + r_period;
double dt = (mat_date - eval_date) / Ntime;

const double dx = (Ar - Al) / (Nspace - 1);
const GridSwing gridswing(Al, Ar, Nspace);
/*  if((Al > 0) || (Ar < 0)) myerror("Error: (Al > 0)  or  (Ar < 0)!");*/

/*if(dx <= 0) myerror("Error: dx = 0!");

  if(dt>dx/(fabs(measure.alpha)+measure.lambda*dx))
  cerr << "Stability Condition is not satisfied!" << endl <<
  "Time Discretization Step is changed" << endl ;*/

while (dt > dx / (fabs(measure.alpha) + measure.lambda * dx))
  {
    Ntime += 10;
    dt = r_period / Ntime;
  }

int i = mat_index;
while (vect_t[i] > mat_date + 0.00000001) i--;

mat_index = i;

const int Kmax = measure.Kmax;

// condition initiale
for (int i = 0; i < Nspace; i++)
  {
    Obst[i] = Price[mat_index][nu][nd][i];
  }

//myerror("Invalid product number");
const int Kmin = measure.Kmin;
//some useful coefficients
//double ss = measure.sigmadiff_squared;

//double dxx = dx*dx;
//double aux = ss/2-(r-divid);
```

```
//vector<double> a(Nspace),b(Nspace),c(Nspace);

/*matrix coefficients of the implicit part*/

double ttm; //time to maturity
for (int n = 0; n < Ntime; n++) //time iterations
  {
    ttm = n * dt;
    /*calculation of the discretized integral using FFT*/
    for (int i = 0; i < Nspace - 1; i++)
      {
        if ((Kmax + 1 + i < 0) || (Kmax + 1 + i >= Nspace))
          {
            uaux[i] = bound_cond_swing(gridswing.x(Kmax + 1 + i), S0, K, rebat

          }
        else uaux[i] = Obst[Kmax + 1 + i];
        uaux_img[i] = 0;
      }
    for (int i = Nspace - 1; i < Nspace + Nz - 1; i++)
      {
        uaux[i] = 0; //zero-padding
        uaux_img[i] = 0;
      }
    for (int i = Nspace + Nz - 1; i < NN_fft; i++)
      {
        if ((Kmin - Nspace - Nz + 1 + i < 0) || (Kmin - Nspace - Nz + 1 + i >=
          {
            uaux[i] = bound_cond_swing(gridswing.x(Kmin - Nspace - Nz + 1 + i)
                                       K, rebate, ttm, r - divid, product, pro
          }
        else uaux[i] = Obst[Kmin - Nspace - Nz + 1 + i];
        uaux_img[i] = 0;
      }

    fft1d(uaux, uaux_img, NN_fft, -1);

    for (int i = 0; i < NN_fft; i++)
      {
        somme[i] = mu[i] * uaux[i] - mu_img[i] * uaux_img[i];
        somme_img[i] = mu_img[i] * uaux[i] + mu[i] * uaux_img[i];
```

```
  }
fft1d(somme, somme_img, NN_fft, 1);

/*computation of the right-hand side vector v */

if (measure.alpha < 0) //backward discretization of the first order deriva
  {
    v[0] = Obst[0] + dt * (somme[NN_fft - 1] - measure.alpha * (Obst[1] -
                           - measure.lambda * Obst[0]) +
           c * dt * bound_cond_swing(gridswing.x(-1), S0, K, rebate, ttm +

    v[Nspace - 1] =  Obst[Nspace - 1] +
                     dt * (somme[Nspace - 2] - measure.alpha * (bound_cond
                           - Obst[Nspace - 1]) / dx - measure.lambda * Obs
                     + b * dt * bound_cond_swing(gridswing.x(Nspace), S0,

    for (int i = 1; i < Nspace - 1; i++)
      v[i] =  Obst[i] + dt * (somme[i - 1] - measure.alpha * (Obst[i + 1]
  }
else  //forward discretization of the first order derivative
  {
    v[0] = Obst [0] + dt * (somme[NN_fft - 1] - measure.alpha * (Obst[0] -
                            bound_cond_swing(gridswing.x(-1), S0, K, rebat
                            - measure.lambda * Obst[0]) +
           c * dt * bound_cond_swing(gridswing.x(-1), S0, K, rebate, ttm +

    for (int i = 1; i < Nspace - 1; i++)
      v[i] =  Obst[i] + dt * (somme[i - 1] - measure.alpha * (Obst[i] - Ob
    v[Nspace - 1] =  Obst[Nspace - 1] + dt * (somme[Nspace - 2] -
                     measure.alpha * (Obst[Nspace - 1]
                                      - Obst[Nspace - 2]) / dx - measure.l
                     + b * dt * bound_cond_swing(gridswing.x(Nspace), S0,
  }

/*computation of de u^(n+1) using LU-decomposition realized in the routine
/*Gauss pivoting*/
Vp[Nspace - 1] = v[Nspace - 1];
beta_p[Nspace - 1] = diag_obst[Nspace - 1];

for (int i = Nspace - 2; i >= 0; i--)
  {
```

```
          beta_p[i] = diag_obst[i] - up_obst[i] * low_obst[i + 1] / beta_p[i + 1
          Vp[i] = v[i] - up_obst[i] * Vp[i + 1] / beta_p[i + 1];
        }

      Obst[0] = Vp[0] / beta_p[0];

      for (int i = 1; i < Nspace; i++)
        {
          Obst[i] = (Vp[i] - low_obst[i] * Obst[i - 1]) / beta_p[i];
        }

    }//end of time iterations
  return 1;
}//end Obst


int price2_swing(int am, const Levy_measure &measure, int product,
                 const int product_type, double r, double divid,
                 double S0, double K, double rebate, double Al,
                 double Ar, int Nspace, double T, int Ntime, int Nu, int Nd, dou
                 double &price0, double &delta0)
{
  double dt = T / Ntime;

  if ((Al > 0) || (Ar < 0)) myerror("Error: (Al > 0)  or  (Ar < 0)!");
  const double dx = (Ar - Al) / (Nspace - 1);
  if (dx <= 0) myerror("Error: dx = 0!");

  if (dt > dx / (fabs(measure.alpha) + measure.lambda * dx))
    cerr << "Stability Condition is not satisfied!" << endl <<
         "Time Discretization Step is changed" << endl ;

  while (dt > dx / (fabs(measure.alpha) + measure.lambda * dx))
    {
      Ntime += 10;
      dt = T / Ntime;
    }

  const int Kmax = measure.Kmax;
  const GridSwing gridswing(Al, Ar, Nspace);
  vector<double>  vect_t(Ntime + 1);
```

```
//vector<double> u(Nspace), v(Nspace);


/*First index where i*t<t-r_period*/
for (int n = 0; n <= Ntime; n++)
  vect_t[n] = n * dt;
int M = Ntime;

mat_index = M;
int i = Ntime;

i = Ntime;
do
  {
    i--;
  }
while (vect_t[i] > (T - r_period) + 0.00000001);

int r_index = i;

/*Memory allocation*/
Price = new double ***[M + 1]; // (double ****)malloc((M+1)*sizeof(double***))

for (int i = 0; i <= M; i++)
  Price[i] = new double **[Nu + 1]; //  (double ***)malloc((Nu+1)*sizeof(doubl

for (int i = 0; i <= M; i++)
  for (int j = 0; j <= Nu; j++)
    Price[i][j] = new double*[Nd + 1]; //  (double **)malloc((Nd+1)*sizeof(dou

for (int i = 0; i <= M; i++)
  for (int j = 0; j <= Nu; j++)
    for (int kk = 0; kk <= Nd; kk++)
      Price[i][j][kk] = new double[Nspace + 1]; // (double *)malloc((N+1)*size

/*Terminal Condition*/
int nu, nd;

for (nu = Nu; nu >= 0; nu--)
  for (nd = Nd; nd >= 0; nd--)
    {
```

```
      if ((nu != 0) || (nd != 0))
        {
          if ((nu != 0) && (nd != 0))
            {
              for (int i = 0; i < Nspace; i++)
                Price[Ntime][nu][nd][i] = fabs(S0 * exp(gridswing.x(i)) - K);
            }
          else if (nu == 0)
            {
              for (int i = 0; i < Nspace; i++)
                Price[Ntime][nu][nd][i] = MAX(0., K - S0 * exp(gridswing.x(i))
            }
          else if (nd == 0)
            {
              for (int i = 0; i < Nspace; i++)
                {
                  Price[Ntime][nu][nd][i] = MAX(0., S0 * exp(gridswing.x(i)) -
                }
            }
        }
      else
        for (int i = 0; i < Nspace; i++)
          {
            Price[Ntime][nu][nd][i] = 0.;
          }
    }


//myerror("Invalid product number");
const int Kmin = measure.Kmin;
//some useful coefficients
double ss = measure.sigmadiff_squared;

//double dxx = dx*dx;
double aux = ss / 2 - (r - divid);

/*matrix coefficients of the implicit part*/
low = new double[Nspace + 1];
diag = new double[Nspace + 1];
up = new double[Nspace + 1];
low_obst = new double[Nspace + 1];
```

```
diag_obst = new double[Nspace + 1];
up_obst = new double[Nspace + 1];

Obst = new double[Nspace + 1];
Vp = new double[Nspace + 1];
beta_p = new double[Nspace + 1];
u = new double[Nspace + 1];
v = new double[Nspace + 1];
init_matrix(dx, dt, ss, aux, r, Nspace);

double dt_obst = r_period / Ntime;
init_matrix_obst(dx, dt_obst, ss, aux, r, Nspace);

double ttm; //time to maturity
int TimeIndex;


N_fft = Nspace + Kmax - Kmin; //number of non-zero values of u involved in com
//zero-padding to obtain NN = N + Nz = 2^p
p = 1;
NN_fft = 2; //size of auxiliary vectors
while (NN_fft < N_fft)
  {
    p++;
    NN_fft = 2 * NN_fft;
  }
Nz = NN_fft - N_fft; // number of extra zeros

mu = new double [NN_fft];
mu_img = new double [NN_fft];
uaux = new double [NN_fft];
uaux_img = new double [NN_fft];
somme = new double [NN_fft];
somme_img = new double [NN_fft];

/*computation of the Fourier transform of nu*/
for (int i = 0; i < Kmax - Kmin + 1; i++)
  {
    mu[i] = (*measure.nu_array)[Kmax - Kmin - i];
    mu_img[i] = 0;
  }
```

```
for (int i = Kmax - Kmin + 1; i < NN_fft; i++)
  {
    mu[i] = 0;
    mu_img[i] = 0;
  }
fft1d(mu, mu_img, NN_fft, -1);


for (int n = Ntime; n >= 1; n--) //time iterations
  {
    ttm = n * dt;
    TimeIndex = n;

    for (nu = 0; nu <= Nu; nu++)
      for (nd = 0; nd <= Nd; nd++)
        if ((nu != 0) || (nd != 0))
          {

            /*calculation of the discretized integral using FFT*/
            for (int i = 0; i < Nspace - 1; i++)
              {
                if ((Kmax + 1 + i < 0) || (Kmax + 1 + i >= Nspace))
                  {
                    uaux[i] = bound_cond_swing(gridswing.x(Kmax + 1 + i), S0,

                  }
                else uaux[i] = Price[TimeIndex][nu][nd][Kmax + 1 + i];
                uaux_img[i] = 0;
              }
            for (int i = Nspace - 1; i < Nspace + Nz - 1; i++)
              {
                uaux[i] = 0; //zero-padding
                uaux_img[i] = 0;
              }
            for (int i = Nspace + Nz - 1; i < NN_fft; i++)
              {
                if ((Kmin - Nspace - Nz + 1 + i < 0) || (Kmin - Nspace - Nz +
                  {
                    uaux[i] = bound_cond_swing(gridswing.x(Kmin - Nspace - Nz
                                               K, rebate, ttm, r - divid, prod
                  }
```

```
    else uaux[i] = Price[TimeIndex][nu][nd][Kmin - Nspace - Nz + 1
    uaux_img[i] = 0;
  }

fft1d(uaux, uaux_img, NN_fft, -1);

for (int i = 0; i < NN_fft; i++)
  {
    somme[i] = mu[i] * uaux[i] - mu_img[i] * uaux_img[i];
    somme_img[i] = mu_img[i] * uaux[i] + mu[i] * uaux_img[i];
  }
fft1d(somme, somme_img, NN_fft, 1);

/*computation of the right-hand side vector v */

if (measure.alpha < 0) //backward discretization of the first orde
  {
    v[0] = Price[TimeIndex][nu][nd][0] + dt * (somme[NN_fft - 1] -
           - measure.lambda * Price[TimeIndex][nu][nd][0]) +
           c * dt * bound_cond_swing(gridswing.x(-1), S0, K, rebat

    v[Nspace - 1] =  Price[TimeIndex][nu][nd][Nspace - 1] +
                     dt * (somme[Nspace - 2] - measure.alpha * (bo
                           - Price[TimeIndex][nu][nd][Nspace - 1])
                     + b * dt * bound_cond_swing(gridswing.x(Nspac

    for (int i = 1; i < Nspace - 1; i++)
      v[i] =  Price[TimeIndex][nu][nd][i] + dt * (somme[i - 1] - m
  }
else  //forward discretization of the first order derivative
  {
    v[0] = Price[TimeIndex][nu][nd] [0] + dt * (somme[NN_fft - 1]
           bound_cond_swing(gridswing.x(-1), S0, K, rebate, ttm, r
           - measure.lambda * Price[TimeIndex][nu][nd][0]) +
           c * dt * bound_cond_swing(gridswing.x(-1), S0, K, rebat

    for (int i = 1; i < Nspace - 1; i++)
      v[i] =  Price[TimeIndex][nu][nd][i] + dt * (somme[i - 1] - m
    v[Nspace - 1] =  Price[TimeIndex][nu][nd][Nspace - 1] + dt * (
                     measure.alpha * (Price[TimeIndex][nu][nd][Nsp
                                      - Price[TimeIndex][nu][nd][N
```

```
                                + b * dt * bound_cond_swing(gridswing.x(Nspac
          }

        /*Gauss pivoting*/
        Vp[Nspace - 1] = v[Nspace - 1];
        beta_p[Nspace - 1] = diag[Nspace - 1];

        for (int i = Nspace - 2; i >= 0; i--)
          {
            beta_p[i] = diag[i] - up[i] * low[i + 1] / beta_p[i + 1];
            Vp[i] = v[i] - up[i] * Vp[i + 1] / beta_p[i + 1];
          }

        Price[TimeIndex - 1][nu][nd][0] = Vp[0] / beta_p[0];

        for (int i = 1; i < Nspace; i++)
          {
            Price[TimeIndex - 1][nu][nd][i] = (Vp[i] - low[i] * Price[Time
          }
      }

if (TimeIndex > r_index)
  {
    for (nu = 0; nu <= Nu; nu++)
      for (nd = 0; nd <= Nd; nd++)
        if ((nu != 0) || (nd != 0))
          {
            if ((nu != 0) && (nd != 0))
              for (int i = 0; i < Nspace; i++)
                Price[TimeIndex - 1][nu][nd][i] = MAX(Price[TimeIndex - 1]
            else if (nu == 0)
              for (int i = 0; i < Nspace; i++)
                Price[TimeIndex - 1][nu][nd][i] = MAX(Price[TimeIndex - 1]
            else if (nd == 0)
              for (int i = 0; i < Nspace; i++)
                Price[TimeIndex - 1][nu][nd][i] = MAX(Price[TimeIndex - 1]
          }

  }
else
  {
```

```
        for (nu = 0; nu <= Nu; nu++)
          for (nd = 0; nd <= Nd; nd++)
            {
              if ((nu != 0) || (nd != 0))
                {
                  if ((nu != 0) && (nd != 0))
                    {
                      /*Call case*/
                      compute_obst(measure, 1, 1, r, divid, S0, K, rebate, Al,
                      for (int i = 0; i < Nspace; i++)
                        Price[TimeIndex - 1][nu][nd][i] = MAX(Price[TimeIndex
                      /*Put case*/
                      compute_obst(measure, 1, 1, r, divid, S0, K, rebate, Al,
                      for (int i = 0; i < Nspace; i++)
                        Price[TimeIndex - 1][nu][nd][i] = MAX(Price[TimeIndex
                    }
                  else if (nu == 0) /*Put case*/
                    {
                      compute_obst(measure, 1, 1, r, divid, S0, K, rebate, Al,
                      for (int i = 0; i < Nspace; i++)
                        Price[TimeIndex - 1][nu][nd][i] = MAX(Price[TimeIndex
                    }
                  else if (nd == 0) /*Call case*/
                    {
                      compute_obst(measure, 1, 1, r, divid, S0, K, rebate, Al,
                      for (int i = 0; i < Nspace; i++)
                        Price[TimeIndex - 1][nu][nd][i] = MAX(Price[TimeIndex
                    }
                }
            }
        }
      //end of time iterations exp(r*(ttm+dt))
    }

int N0 = (int) floor(-Al / dx);
double Sl = S0 * exp(gridswing.x(N0 - 1));
double Sm = S0 * exp(gridswing.x(N0));
double Sr = S0 * exp(gridswing.x(N0 + 1));

// S0 is between Sm and Sr
double pricel = Price[0][Nu][Nd][N0 - 1];
```

```
double pricem = Price[0][Nu][Nd][N0];
double pricer = Price[0][Nu][Nd][N0 + 1];

//quadratic interpolation
double A = pricel;
double B = (pricem - pricel) / (Sm - Sl);
double C = (pricer - A - B * (Sr - Sl)) / (Sr - Sl) / (Sr - Sm);
price0 = A + B * (S0 - Sl) + C * (S0 - Sl) * (S0 - Sm);
delta0 = B + C * (2 * S0 - Sl - Sm);

delete [] low;
delete [] diag;
delete [] up;
delete [] low_obst;
delete [] diag_obst;
delete [] up_obst;

delete [] Obst;
delete [] Vp;
delete [] beta_p;
delete [] u;
delete [] v;

delete [] mu;
delete [] mu_img;
delete [] uaux;
delete [] uaux_img;
delete [] somme;
delete [] somme_img;

int j, kk;

for (i = 0; i <= M; i++)
  for (j = 0; j <= Nu; j++)
    for (kk = 0; kk <= Nd; kk++)
      delete[] Price[i][j][kk];

for (i = 0; i <= M; i++)
  for (j = 0; j <= Nu; j++)
    delete[] Price[i][j];
```

```
  for (i = 0; i <= M; i++)
    delete[] Price[i];

  delete [] Price;


  return 1;
}//end price2_swing

#endif //PremiaCurrentVersion
```