Help

```
/*
 * American option pricing with the underlying asset following a Samuelson
 * dynamics in two dimensions using the methodology of:
 *
 * Barty, K., Roy, J.-S., and Strugarek, C. (2005).
 * Temporal difference learning with kernels.
 * Available at Optimization Online:
 * http://www.optimization-online.org/DB_HTML/2005/05/1133.html.
 *
 * with enhancements by Girardeau, P.
 *
 * More information on the specifics of the implemetation can be found in the
 * accompagnying documentation.
 *
 * The code was written by Girardeau, P. and Roy, J.-S. at the EDF R&D and is
 * Copyright (c) 2005-2006, EDF SA.
 */

#include <cstdlib>
#include <iostream>
#include <cmath>
#include <vector>

using namespace std;

extern "C" {
#include  "bs2d_std2d.h"
#include "enums.h"
}
#include "pnl/pnl_mathtools.h"

/* Type definitions */

typedef struct ifgt_set_
{
  double *C; /* coefficients of the Taylor expansion : C[box*binom+index] */
  int Kd; /* number of centers (number of boxes per dimension) */
  int binom; /* number of coefficients */
  int d; /* state dimension */
```

```
  int p; /* degree of the Taylor expansion */
  int rho; /* ~ number of neighbours to be considered */
  double h; /* bandwidth */
} ifgt_set;

typedef struct liste_ifgt_
{
  ifgt_set f;
  struct liste_ifgt_ *next;
} liste_ifgt;

typedef struct ifgt_
{
  int p; /* degree of the Taylor expansion */
  int rho; /* ~ number of neighbours to be considered */
  int d; /* state dimension */
  struct liste_ifgt_ *liste; /* 1st element of the list */
  double h0; /* first bandwidth, next ones decrease like h0*2^i */
} ifgt;


/* Prototypes */



static int nchoosek(int n, int k);
static void ifgt_set_init(ifgt_set *f, int d, int p, int rho, double h);
static void ifgt_set_add(ifgt_set *f, std::vector<double> &x, double q);
static double ifgt_set_eval(ifgt_set *f, std::vector<double> &x);
static void ifgt_init(ifgt *F);
static void ifgt_add(ifgt *F, std::vector<double> &x, double q, double h);
static double ifgt_eval(ifgt *F, std::vector<double> &x);
static void ifgt_free(ifgt *F);

static void alea_bb_traj(std::vector<double *> &x, double *x0, double dt, double
                         double *si, double r, double *divid, int generator, int


static int MC_BGRS2D_aux(double x01, double x02, NumFunc_2  *p, double tmax,
                         double r, double divid1, double divid2, double sigma1,
                         double rho, long N, int generator, double inc, int exer
```

```
                              double *ptprice, double *ptdelta1, double *ptdelta2);


/* IFGT toolbox in [0, 1]**2 */

int nchoosek(int n, int k)
{
  int n_k = n - k;
  int nchsk = 1;
  int i;

  if (k < n_k)
    {
      k = n_k;
      n_k = n - k;
    }

  for (i = 1; i <= n_k; i++)
    nchsk = (nchsk * (++k)) / i;

  return nchsk;
}

void ifgt_set_init(ifgt_set *f, int d, int p, int rho, double h)
{
  int i, K;
  f->Kd = (int)ceil(0.5 / h);

  for (K = f->Kd, i = 1; i < d; i++) K *= f->Kd; /* Kd ^ d */
  f->p = p;
  f->rho = rho;
  f->d = d;
  f->h = h;

  /* Initialization of C to 0 */
  f->binom = nchoosek(p + d, d);
  f->C = (double *)calloc(K * f->binom, sizeof(*(f->C)));
}

void ifgt_set_add(ifgt_set *f, std::vector<double> &x, double q)
{
```

```
std::vector<int> heads(f->d + 1);
int k, i, j, t, tail, ind;
std::vector<int> cinds(f->binom);
std::vector<double> dx(f->d);
std::vector<double> ck(f->d);
std::vector<double> prods(f->binom);
double sum, sum2, *v;

/* find the nearest center (ck) from x */
for (ind = 0, i = 0; i < f->d; i++)
  {
    if (x[i] < 0. || x[i] > 1.) return;

    j = (int)floor(x[i] * 0.5 / f->h);
    ind = ind * f->Kd + j;
    ck[i] = (j + 0.5) * f->h * 2;
  }

/* compute dx */
sum = 0.0;
for (i = 0; i < f->d; i++)
  {
    dx[i] = (x[i] - ck[i]) / f->h;
    sum -= dx[i] * dx[i];
    heads[i] = 0;
  }
heads[f->d] = f->binom + 1;
sum2 = q * exp(sum);

/* for factorial(alpha) */
cinds[0] = 0;

/* update the coefficients with the new kernel */
prods[0] = 1.0;
v = &f->C[ind * f->binom];
v[0] += sum2;

/* recursive computing of multinomes (inspired by Yang) */
for (k = 1, t = 1, tail = 1; k < f->p; k++, tail = t) /* boucle sur les puissa
  for (i = 0; i < f->d; i++) /* boucle sur les coordonnées */
    for (j = heads[i], heads[i] = t; j < tail; j++, t++)
```

```
        {
          /* for factorial(alpha) */
          cinds[t] = (j < heads[i + 1]) ? cinds[j] + 1 : 1;
          /* compute powers */
          prods[t] = dx[i] * 2.0 * prods[j] / cinds[t];

          v[t] += sum2 * prods[t];
        }
}

double ifgt_set_eval(ifgt_set *f, std::vector<double> &x)
{
  int sfac = 2 * (f->rho) + 1, b0, b1, j0, j1, b0min, b0max, b1min, b1max;
  double res = 0.0, d0, d1, *v, inv = 1.0 / f->h, d1b;

  b0 = ((int)floor(x[0] * inv * 0.5)) - f->rho;
  b0min = b0 > 0 ? b0 : 0;
  b0max = b0 + sfac > f->Kd ? f->Kd : b0 + sfac;
  d0 = x[0] * inv - 1 - 2 * b0min;

  b1 = ((int)floor(x[1] * inv * 0.5)) - f->rho;
  b1min = b1 > 0 ? b1 : 0;
  b1max = b1 + sfac > f->Kd ? f->Kd : b1 + sfac;
  d1b = x[1] * inv - 1 - 2 * b1min;

  /* for every box near the one containing x */
  for (j0 = b0min; j0 < b0max; j0++, d0 -= 2)
    for (d1 = d1b, j1 = b1min; j1 < b1max; j1++, d1 -= 2)
      {
        v = &f->C[(j0 * f->Kd + j1) * f->binom];
        res += ((((v[14] * d1 + (v[13] * d0 + v[9])) * d1 + (v[12] * d0 + v[8])
                 + ((v[11] * d0 + v[7]) * d0 + v[4]) * d0 + v[2]) * d1
                + (((v[10] * d0 + v[6]) * d0 + v[3]) * d0 + v[1]) * d0
                + v[0]) * exp(-(d0 * d0 + d1 * d1));
      }

  return res;
}

void ifgt_init(ifgt *F)
{
```

```
  F->d = 2;
  F->liste = NULL;
  /* Default values for approx. 0.001 rel. precision */
  F->rho = 1;
  F->p = 5; /* DO NOT CHANGE THIS unless you change ifgt_set_eval */
}

void ifgt_add(ifgt *F, std::vector<double> &x, double q, double h)
{
  liste_ifgt *Ltmp, *Ltmp2 = NULL;

  if (F->liste == NULL)
    F->h0 = h;

  /* find the floor with f.h the nearest from h */
  for (Ltmp = F->liste; Ltmp != NULL; Ltmp2 = Ltmp, Ltmp = Ltmp->next)
    if (Ltmp->f.h * .5 < h && h <= Ltmp->f.h) break;

  if (Ltmp == NULL) /* if we did not find a "good h" */
    {
      /* compute the nearest h0*2^i from h */
      double htmp = F->h0 * pow(2.0 , ceil(log(h / F->h0) / log(2.0)));

      Ltmp = (liste_ifgt *) malloc(sizeof(*Ltmp));

      /* create a new floor */
      /* pointer to the next : NULL */
      Ltmp->next = NULL;

      /* Initialization of the corresponding fgt_set */
      ifgt_set_init(&(Ltmp->f), F->d, F->p, F->rho, htmp);

      if (F->liste) /* if F->liste is not NULL */
        Ltmp2->next = Ltmp; /* put it behind */
      else /* else */
        F->liste = Ltmp;
    }

  /* add x to the floor */
  ifgt_set_add(&(Ltmp->f), x, q);
}
```

```
double ifgt_eval(ifgt *F, std::vector<double> &x)
{
  double res = 0.0;
  liste_ifgt *Ltmp;

  /* Sum over all bandwidths */
  for (Ltmp = F->liste; Ltmp != NULL; Ltmp = Ltmp->next)
    res += ifgt_set_eval(&(Ltmp->f), x);

  return res;
}

void ifgt_free(ifgt *F)
{
  liste_ifgt *Ltmp, *L = F->liste;

  while (L) /* for every non-empty floor */
    {
      Ltmp = L;
      L = L->next;
      free(Ltmp->f.C);
      free(Ltmp);
    }
}

/* Compute price processes following Samuelson dynamic in dim. 2 */

void alea_bb_traj(std::vector<double *> &x, double *x0, double dt, double L[2][2
                  double r, double *divid, int generator, int nmax)
{
  int j, n = pnl_rand_or_quasi(generator);
  double tmax = dt * nmax;
  double W1, W2;

  /* log-tranform */
  for (j = 0; j < 2; j++)
    x[0][j] = log(x0[j]);

  /* draw all the transition noises */
  pnl_rand_gauss(2 * nmax, CREATE, 0, generator);
```

```
  /* draw x(nmax) */
  W1 = pnl_rand_gauss(2 * nmax, RETRIEVE, 0, generator);
  W2 = pnl_rand_gauss(2 * nmax, RETRIEVE, 1, generator);

  for (j = 0; j < 2; j++)
    x[nmax][j] = x[0][j] + ((r - divid[j]) - si[j] * si[j] / 2) * tmax +
                 sqrt(tmax) * (L[j][0] * W1 + L[j][1] * W2);

  /* compute brownian bridge from the end */
  for (n = nmax - 1; n >= 1; n--)
    {
      double t = n * dt;

      W1 = pnl_rand_gauss(2 * nmax, RETRIEVE, 2 * n, generator);
      W2 = pnl_rand_gauss(2 * nmax, RETRIEVE, 2 * n + 1, generator);

      /* dynamic */
      for (j = 0; j < 2; j++)
        x[n][j] = x[0][j] + (t / (t + dt)) * (x[n + 1][j] - x[0][j]) +
                  sqrt(t * dt / (t + dt)) * (L[j][0] * W1 + L[j][1] * W2);
    }

  /* inverse log-transform */
  for (n = 0; n <= nmax; n++)
    for (j = 0; j < 2; j++)
      x[n][j] = exp(x[n][j]);
}


/* Other functions */




/*
 * Main function
 */
int MC_BGRS2D_aux(double x01, double x02, NumFunc_2  *p, double tmax, double r,
                  double divid1, double divid2, double sigma1, double sigma2, do
                  long N, int generator, double inc, int exercise_date_number, d
```

```
                  double *ptdelta1, double *ptdelta2)
{
  double dt = tmax / (exercise_date_number - 1.), exprdt = exp(-r * dt);
  int k, j, n, d = 2, nmax = (int)floor(tmax / dt), k0 = (int)floor(60.0 * N / 1
  /* problem variables */
  double sigma[2], divid[2];
  /* cholesky */
  double L[2][2];
  std::vector<ifgt> f(nmax + 1); /* optimal control for every step n */
  /* price process xi */
  std::vector<double *> xi(nmax + 1);
  double x[5][2];
  /* results */
  double J[5] = {0, 0, 0, 0, 0}, Jmoy[5] = {0, 0, 0, 0, 0};

  /* Initializations */

  sigma[0] = sigma1;
  sigma[1] = sigma2;
  divid[0] = divid1;
  divid[1] = divid2;

  /* covariance of the noises */
  L[0][0] = sigma[0];
  L[0][1] = 0.0;
  L[1][0] = rho * sigma[1];
  L[1][1] = sqrt(1 - rho * rho) * sigma[1];

  /* starting point */
  x[0][0] = x01;
  x[0][1] = x02;
  x[1][0] = x[0][0] - inc * x01;
  x[1][1] = x[0][1];
  x[2][0] = x[0][0] + inc * x01;
  x[2][1] = x[0][1];
  x[3][0] = x[0][0];
  x[3][1] = x[0][1] - inc * x02;
  x[4][0] = x[0][0];
  x[4][1] = x[0][1] + inc * x02;

  /* initialization of the fgt */
```

```
for (n = 0; n <= nmax; n++)
  {
    ifgt_init(&(f[n]));
    xi[n] = (double *)malloc(d * sizeof(double));
  }

/* Test after initialization for the generator */
if (pnl_rand_init(generator, 2 * nmax, N) == OK)
  {
    for (k = 0; k < N; k++)
      {
        std::vector<double> logxi1(d);
        std::vector<double> logxi2(d);
        /* turn over all starting points for hedging computation */
        for (j = 0; j < d; j++)
          xi[0][j] = x[k % 5][j];

        /* draw price process xi */
        alea_bb_traj(xi, xi[0], dt, L, sigma, r, divid, generator, nmax);

        /* update */
        for (n = nmax - 1; n >= 0; n--)
          {
            /* steps of the algorithm */
            double td, rho_pow = 0.3, rho = 1.1 / pow(k + 1.0, rho_pow),
                       eps_pow = 0.2, eps = 1.0 / pow(k + 1.0, eps_pow);

            /* transform lognormal into normal centered on 0.5 */
            for (j = 0; j < d; j++)
              {
                if (n > 0)
                  logxi1[j] =
                    (log(xi[n][j]) - log(x[0][j]) - n * dt
                     * ((r - divid[j]) - sigma[j] * sigma[j] / 2))
                    / (sigma[j] * sqrt(n * dt) * 10.0) + 0.5;
                logxi2[j] =
                  (log(xi[n + 1][j]) - log(x[0][j]) - (n + 1) * dt
                   * ((r - divid[j]) - sigma[j] * sigma[j] / 2))
                  / (sigma[j] * sqrt((n + 1) * dt) * 10.0) + 0.5;
              }
            /* temporal difference */
```

```
            if (n > 0)
              td = exprdt * MAX((p->Compute)(p->Par, xi[n + 1][0], xi[n + 1][1
                                ifgt_eval(&(f[n + 1]), logxi2)) - ifgt_eval(&(
            else
              td = exprdt * MAX((p->Compute)(p->Par, xi[n + 1][0], xi[n + 1][1
                                ifgt_eval(&(f[n + 1]), logxi2)) - J[k % 5];

            /* update fgt */
            if (n > 0)
              ifgt_add(&(f[n]), logxi1, rho * td, eps);
            else
              J[k % 5] += rho * td;
          }

        /* Polyak Juditsky */
        if (k < k0)
          Jmoy[k % 5] = J[k % 5];
        else
          Jmoy[k % 5] += (J[k % 5] - Jmoy[k % 5]) / (k / 5 + 1 - k0 / 5);

      }
    }

  *ptprice = MAX(Jmoy[0], (p->Compute)(p->Par, x[0][0], x[0][1]));
  *ptdelta1 = (MAX(Jmoy[2], (p->Compute)(p->Par, x[2][0], x[2][1]))
               - MAX(Jmoy[1], (p->Compute)(p->Par, x[1][0], x[1][1]))) / (2 * in
  *ptdelta2 = (MAX(Jmoy[4], (p->Compute)(p->Par, x[4][0], x[4][1]))
               - MAX(Jmoy[3], (p->Compute)(p->Par, x[3][0], x[3][1]))) / (2 * in

  /* free memory */
  for (n = 0; n <= nmax; n++)
    {
      ifgt_free(&(f[n]));
      free(xi[n]);
    }

  return 0;
}

extern "C" {
  int CALC(MC_BGRS2D)(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 MC_BGRS2D_aux(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.V_PDOUBLE,
                       ptMod->Rho.Val.V_RGDOUBLE,
                       Met->Par[0].Val.V_LONG,
                       Met->Par[1].Val.V_ENUM.value,
                       Met->Par[2].Val.V_PDOUBLE,
                       Met->Par[3].Val.V_INT,
                       &(Met->Res[0].Val.V_DOUBLE),
                       &(Met->Res[1].Val.V_DOUBLE),
                       &(Met->Res[2].Val.V_DOUBLE));
}

static int CHK_OPT(MC_BGRS2D)(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)
{
  static int first = 1;
```

```
    if (first)
      {
        Met->Par[0].Val.V_LONG = 50000;
        Met->Par[1].Val.V_ENUM.value = 0;
        Met->Par[1].Val.V_ENUM.members = &PremiaEnumRNGs;
        Met->Par[2].Val.V_PDOUBLE = 0.1;
        Met->Par[3].Val.V_INT = 10;
        first = 0;
      }
    return OK;
  }

  PricingMethod MET(MC_BGRS2D) =
  {
    "MC_BartyRoyStrugarek2d",
    { {"N iterations", LONG, {100}, ALLOW},
      {"RandomGenerator", ENUM, {100}, ALLOW},
      {"Delta Increment Rel", PDOUBLE, {100}, ALLOW},
      {"Number of Exercise Dates", INT, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_BGRS2D),
    { {"Price", DOUBLE, {100}, FORBID},
      {"Delta1", DOUBLE, {100}, FORBID} ,
      {"Delta2", DOUBLE, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(MC_BGRS2D),
    CHK_mc,
    MET(Init)
  };
};
```