

## Help

```
/*
 * American option pricing with the underlying asset following a Samuelson
 * dynamics in one dimension 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
 * accompagnyng 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 <
href../../common/math/highdim_solver/highdim_vector_h_src.pdfvector>

using namespace std;

extern "C" {
#include "
href../../mod/bs1d/bs1d_std/bs1d_std_h_src.pdfbs1d_std.h"
#include "
href../../common/enums_h_src.pdfenums.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 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  $h0 \cdot 2^i$  */
} ifgt;

/* Prototypes */

static void ifgt_set_init(ifgt_set *f, int p, int rho, double h);
static void ifgt_set_add(ifgt_set *f, double x, double q);
static double ifgt_set_eval(ifgt_set *f, double x);
static void ifgt_init(ifgt *F);
static void ifgt_add(ifgt *F, double x, double q, double h);
static double ifgt_eval(ifgt *F, double x);
static void ifgt_free(ifgt *F);

static void alea_bb_traj(std::vector<double> &x, double x0, double dt, double si,
                        double divid, int generator, int nmax);

static inline double max(double a, double b);

static int MC_BGRS_aux(double x, NumFunc_1 *p, double tmax, double r,
                      double divid, double sigma, long N, int generator, double
                      int exercise_date_number, double *ptprice, double *ptdelt

```

```

/* IFGT toolbox on [0, 1] */

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

    f->p = p;
    f->rho = rho;
    f->h = h;

    /* Initialization of C to 0 */
    f->C = (double *)calloc(f->Kd * f->p, sizeof(*(f->C)));
}

void ifgt_set_add(ifgt_set *f, double x, double q)
{
    int ind = (int)floor((x / f->h) * 0.5), i;
    double dx = (x / f->h) - (2.0 * ind + 1), puis, fact, sum2, *v;

    if (ind < 0 || ind >= f->Kd) return;

    sum2 = q * exp(-dx * dx);

    /* update the coefficients with the new kernel */
    v = &f->C[ind * f->p];
    v[0] += sum2;
    for (i = 1, puis = 2 * dx, fact = 1; i < f->p; i++, puis *= 2 * dx, fact *= i)
        v[i] += sum2 * puis / fact;
}

double ifgt_set_eval(ifgt_set *f, double x)
{
    int k, b = (int)floor((x / f->h) * 0.5), ind, minind, maxind;
    double *v, rest, res = 0.0, dx;

    minind = b - f->rho < 0 ? 0 : b - f->rho;
    maxind = b + f->rho + 1 < f->Kd ? b + f->rho + 1 : f->Kd;
    /* for every box near the one containing x */
    for (ind = minind, dx = x / f->h - (ind * 2 + 1); ind < maxind; ind++, dx -= 2)
        {

```

```

    v = &f->C[ind * f->p];

    for (rest = v[f->p - 1], k = f->p - 2; k >= 0; k--)
        rest = rest * dx + v[k];

    res += rest * exp(-dx * dx);
}

return res;
}

void ifgt_init(ifgt *F)
{
    F->liste = NULL;
    /* Default values for 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, 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->p, F->rho, htmp);

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

/* ajout de x a l'etage */
ifgt_set_add(&(Ltmp->f), x, q);
}

double ifgt_eval(ifgt *F, 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. 1 */

void alea_bb_traj(std::vector<double> &x, double x0, double dt, double si, doubl

```

```

double divid, int generator, int nmax)
{
    int n = pnl_rand_or_quasi(generator);
    double tmax = dt * nmax, W, l0;

    /* log-tranform */
    l0 = log(x0);

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

    /* draw x(nmax) */
    W = pnl_rand_gauss(nmax, RETRIEVE, 0, generator);
    x[nmax] = l0 + ((r - divid) - si * si / 2) * tmax + sqrt(tmax) * si * W;

    /* compute brownian bridge from the end */
    for (n = nmax - 1; n >= 1; n--)
    {
        double t = n * dt;
        W = pnl_rand_gauss(nmax, RETRIEVE, n, generator);
        /* dynamic */
        x[n] = l0 + (t / (t + dt)) * (x[n + 1] - l0) + sqrt(t / (n + 1)) * si * W;
    }

    /* inverse log-transform */
    for (n = 1; n <= nmax; n++)
        x[n] = exp(x[n]);
}

/* Other functions */

inline double max(double a, double b)
{
    return (a > b) ? a : b;
}

/*
 * Main function
 */
int MC_BGRS_aux(double x, NumFunc_1 *p, double tmax, double r, double divid,
                double sigma, long N, int generator, double inc, int exercise_da

```

```

double *ptprice, double *ptdelta)
{
    double dt = tmax / (exercise_date_number - 1.), exprdt = exp(-r * dt);
    int k, n, k0 = (int)floor(60.0 * N / 100), nmax = (int)floor(tmax / dt);
    std::vector<ifgt> f(nmax + 1); /* optimal control for every step n */
    /* price process xi */
    std::vector<double> xi(nmax + 1);
    /* Results */
    double J[3] = {0, 0, 0}, Jmoy[3] = {0, 0, 0};

    /* initialization of the fgt */
    for (n = 0; n <= nmax; n++)
        ifgt_init(&(f[n]));

    /* Test after initialization for the generator */
    if (pnl_rand_init(generator, nmax, N) == OK)
    {
        for (k = 0; k < N; k++)
        {
            /* add increment for hedging computation */
            xi[0] = x + ((k % 3) - 1) * inc * x;

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

            /* update */
            for (n = nmax - 1; n >= 0; n--)
            {
                /* steps of the algorithm */
                double rho_pow = 0.3;
                double rho = 1.1 / pow(k + 1.0, rho_pow);
                double eps_pow = 0.3;
                double eps = 1.0 / pow(k + 1.0, eps_pow);
                double td, logxi1 = 0.0, logxi2;

                /* transform lognormal into normal centered on 0.5 */
                if (n > 0)
                {
                    logxi1 = (log(xi[n]) - log(x) - n * dt * (r - sigma * sigma / 2)) /
                        (rho * eps);
                    logxi2 = (log(xi[n + 1]) - log(x) - (n + 1) * dt * (r - sigma * sigma / 2)) /
                        (rho * eps);

                    /* temporal difference */

```

```

        if (n > 0)
            td = exprdt * max(p->Compute(p->Par, xi[n + 1]),
                             ifgt_eval(&(f[n + 1]), logxi2)) - ifgt_eval(&(
        else
            td = exprdt * max((p->Compute)(p->Par, xi[n + 1]),
                             ifgt_eval(&(f[n + 1]), logxi2)) - J[k % 3];

        /* update fgt */
        if (n > 0)
            ifgt_add(&(f[n]), logxi1, rho * td, eps);
        else
            J[k % 3] += rho * td;
    }

    /* Polyak Juditsky */
    if (k < k0)
        Jmoy[k % 3] = J[k % 3];
    else
        Jmoy[k % 3] += (J[k % 3] - Jmoy[k % 3]) / (k / 3 + 1 - k0 / 3);
}

}

*ptprice = max(Jmoy[1], p->Compute(p->Par, x));
*ptdelta = (max(Jmoy[2], p->Compute(p->Par, x + inc * x)) -
            max(Jmoy[0], p->Compute(p->Par, x - inc * x))) / (2 * x * inc);

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

return 0;
}
extern "C" {
    int CALC(MC_BGRS)(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 MC_BGRS_aux(ptMod->S0.Val.V_PDOUBLE,
                  ptOpt->PayOff.Val.V_NUMFUNC_1,
                  ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                  r,
                  divid,
                  ptMod->Sigma.Val.V_PDOUBLE,
                  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));
}

static int CHK_OPT(MC_BGRS)(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 *Mod)
{
    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 = 0.01;
        Met->Par[3].Val.V_INT = 10;
        first = 0;
    }

    return OK;
}

```

```

PricingMethod MET(MC_BGRS) =
{
    "MC_BartyRoyStrugarek",
    { {"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_BGRS),
    { {"Price", DOUBLE, {100}, FORBID},
      {"Delta", DOUBLE, {100}, FORBID} ,
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(MC_BGRS),
    CHK_mc,
    MET(Init)
};
}

```