

Help

```

#include "merhes1d_std.h"
#include "pnl/pnl_mathtools.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
static int CHK_OPT(FD_MertonHeston)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(FD_MertonHeston)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

#include "math/highdim_solver/cps_function.h"
#include "math/highdim_solver/cps_pde.h"
#include "math/highdim_solver/cps_pde_term.h"
#include "math/highdim_solver/cps_pde_integral_term.h"
#include "math/highdim_solver/cps_pde_problem.h"
#include "math/highdim_solver/cps_grid.h"
#include "math/highdim_solver/cps_grid_node.h"
#include "math/highdim_solver/cps_grid_tuner.h"
#include "math/highdim_solver/cps_stencil_operator.h"
#include "math/highdim_solver/cps_boundary_description.h"
#include "math/highdim_solver/cps_assertions.h"
#include "math/highdim_solver/cps_debug.h"
#include "math/highdim_solver/cps_utils.h"

typedef struct bates_model_t
{
    double T;
    double theta, sigma, delta, r, rho, Ks;
    double E, K;
    double S0, V0, X0, Y0;
    double alpha, lambda, m;
    unsigned int Ns, Nv;
} bates_model;

```

```
/* functions */

double cps_func_zero(const function *f, const grid_node *gn)
{

    REQUIRE("function_not_null", f != NULL);
    REQUIRE("grid_node_not_null", gn != NULL);

    return 0.0;
}

/* public methods */

static int cps_function_create(function **f)
{

    STANDARD_CREATE(f, function);
    (*f)->body = cps_func_zero;
    return OK;
}

static int cps_function_destroy(function **f)
{

    STANDARD_DESTROY(f);
    return OK;
}

static int cps_function_set_body(function *f, double (*body)(const function *, c
{
    /* set body of function */
    REQUIRE("function_not_null", f != NULL);
    REQUIRE("body_not_null", body != NULL);

    f->body = body;

    return OK;
}

static int cps_function_set_args(function *f, const void *args)
{
```

```

/* set function arguments */
REQUIRE("function_not_null", f != NULL);
REQUIRE("args_not_null", args != NULL);

f->args = args;
return OK;
}

/* tuning functions */

static int focus_rescaler_proc(grid_tuner *tuner, grid *grid)
{

    bates_model *m;
    /* rescale grid around focus */
    REQUIRE("tuner_not_null", tuner != NULL);
    REQUIRE("grid_not_null", grid != NULL);
    m = (bates_model *)tuner->argument;

    if (!(grid->ticks[X_DIM] % 2))
        grid->ticks[X_DIM]++;
    if (!(grid->ticks[Y_DIM] % 2))
        grid->ticks[Y_DIM]++;

    grid->min_value[X_DIM] = 0.0;
    grid->max_value[X_DIM] = 1.;
    grid->min_value[Y_DIM] = 0.0;
    grid->max_value[Y_DIM] = grid->min_value[Y_DIM] + 8.0 * m->V0;

    grid->focus[X_DIM] = 0.5;

    grid->focus_tick[Y_DIM] = (int)floor((m->V0 - grid->min_value[Y_DIM]) * (grid->ticks[Y_DIM] - 1));

    grid->max_value[Y_DIM] = grid->min_value[Y_DIM] + ((double)(grid->ticks[Y_DIM] - 1) * (m->V0 - grid->min_value[Y_DIM]) / (grid->focus_tick[Y_DIM] - grid->min_value[Y_DIM]));

    grid->delta[X_DIM] = (grid->max_value[X_DIM] - grid->min_value[X_DIM]) / ((double)(grid->ticks[X_DIM] - 1));
    grid->delta[Y_DIM] = (grid->max_value[Y_DIM] - grid->min_value[Y_DIM]) / ((double)(grid->ticks[Y_DIM] - 1));

    grid->focus[Y_DIM] = grid->min_value[Y_DIM] + ((double)grid->focus_tick[Y_DIM] * (grid->delta[Y_DIM] - grid->min_value[Y_DIM]) / (grid->focus_tick[Y_DIM] - grid->min_value[Y_DIM]));
}

```

```

grid->focus_tick[X_DIM] = (int)floor(0.5 * (grid->ticks[X_DIM]));

//xmin = grid->min_value[X_DIM];
//ymin = grid->min_value[Y_DIM];
//xmax = grid->min_value[X_DIM] + ((double)(grid->ticks[X_DIM] - 1)) * grid->d
//ymax = grid->min_value[Y_DIM] + ((double)(grid->ticks[Y_DIM] - 1)) * grid->d

return OK;
}

static int explicit_tuner_proc(grid_tuner *tuner, grid *grid)
{
    /* tuning procedure for explicit part */

    bates_model *model = (bates_model *)tuner->argument;

    double dx ;
    double rx = 0.03125;

    double dy;
    double ry = 0.5;

    double bx ;
    double by;
    dx = grid->delta[X_DIM];
    dy = grid->delta[Y_DIM];
    bx = POW(dx, 2.0) / (0.5 * rx);
    by = POW(dy, 2.0) / (0.5 * ry * POW(model->sigma, 2.0));
    REQUIRE("tuner_not_null", tuner != NULL);
    REQUIRE("grid_not_null", grid != NULL);
    REQUIRE("grid_is_rescaled", grid->is_rescaled);
    grid->delta[T_DIM] = 0.1 * MIN(bx, by);

    grid->ticks[T_DIM] = (int)floor((grid->max_value[T_DIM] - grid->min_value[T_DIM]
    grid->delta[T_DIM] = (grid->max_value[T_DIM] - grid->min_value[T_DIM]) / ((double)grid->ticks[T_DIM] - 1));

    ENSURE("tmax_accurate", APPROX_EQUAL(grid->max_value[T_DIM],
                                           (grid->min_value[T_DIM] + ((double)grid->ticks[T_DIM] - 1) * grid->delta[T_DIM]));

    return OK;
}

```

```

}

static int implicit_tuner_proc(grid_tuner *tuner, grid *grid)
{
    /* tuning procedure for implicit part */
    REQUIRE("tuner_not_null", tuner != NULL);
    REQUIRE("grid_not_null", grid != NULL);

    grid->current_value[T_DIM] -= grid->delta[T_DIM]; /* return to last step compu
    grid->delta[T_DIM] = sqrt(grid->delta[T_DIM]);          /* rescale dt */

    grid->delta[T_DIM] = MIN(grid->delta[X_DIM], grid->delta[Y_DIM]);

    grid->min_value[T_DIM] = grid->current_value[T_DIM] + grid->delta[T_DIM]; /* c

    grid->ticks[T_DIM] = (unsigned int)floor((grid->max_value[T_DIM] - grid->min_v
    grid->delta[T_DIM] = (grid->max_value[T_DIM] - grid->min_value[T_DIM]) / ((dou

    ENSURE("tmax_accurate", APPROX_EQUAL(grid->max_value[T_DIM],
                                          (grid->min_value[T_DIM] + ((double)grid->

    return OK;
}

/* model functions */

static double func_call_payoff(const function *f, const grid_node *node)
{
    bates_model *model = (bates_model *)f->args;

    double x = node->value[X_DIM];
    double K = model->K;
    double S0 = model->S0;

    double result = MAX(x * (S0 + K) / S0 - K / S0, 0.);

    REQUIRE("function_not_null", f != NULL);
    REQUIRE("node_not_null", node != NULL);

    return result;
}

```

```
}
```

```
/*
```

```
static double func_put_payoff(const function *f, const grid_node *node){
```

```
    REQUIRE("function_not_null", f != NULL);
```

```
    REQUIRE("node_not_null", node != NULL);
```

```
    bates_model *model = (bates_model *)f->args;
```

```
    double x = node->value[X_DIM];
```

```
    double K = model->K;
```

```
    double S0 = model->S0;
```

```
    double result = max((K/S0 * (1.0 - x) - x), 0.0);
```

```
    return result;
```

```
}
```

```
*/
```

```
static double func_call_boundary(const function *f, const grid_node *node)
```

```
{
```

```
    bates_model *model = (bates_model *)f->args;
```

```
    double x = node->value[X_DIM];
```

```
    double t = node->value[T_DIM];
```

```
    double K = model->K;
```

```
    double r = model->r;
```

```
    double S0 = model->S0;
```

```
    double result = MAX(x * (S0 + K * exp(-r * t)) / S0 - K * exp(-r * t) / S0, 0.
```

```
    REQUIRE("function_not_null", f != NULL);
```

```
    REQUIRE("node_not_null", node != NULL);
```

```
    return result;
```

```
}
```

```
/*
```

```
static double func_put_boundary(const function *f, const grid_node *node){
```

```
    REQUIRE("function_not_null", f != NULL);
```

```
    REQUIRE("node_not_null", node != NULL);

    bates_model *model = (bates_model *)f->args;

    double x = node->value[X_DIM];
    double t = node->value[T_DIM];
    double K = model->K;
    double r = model->r;
    double S0 = model->S0;

    double result = max(K * exp(-r*t)*(1.0 - x)/S0 - x, 0.0);

    return result;
}
*/

static double func_uxx(const function *f, const grid_node *node)
{

    double x = node->value[X_DIM];
    double y = node->value[Y_DIM];

    double result = 0.5 * y * POW(x * (1.0 - x), 2.0);
    REQUIRE("function_not_null", f != NULL);
    REQUIRE("node_not_null", node != NULL);
    return result;
}

static double func_uxy(const function *f, const grid_node *node)
{

    bates_model *model = (bates_model *)f->args;

    double x = node->value[X_DIM];
    double y = node->value[Y_DIM];

    double rho = model->rho;
    double sigma = model->sigma;
```

```
double result = rho * sigma * y * x * (1.0 - x);
    REQUIRE("function_not_null", f != NULL);
    REQUIRE("node_not_null", node != NULL);
    return result;
}

static double func_uyy(const function *f, const grid_node *node)
{
    bates_model *model;
    double y ;
    double sigma;
    double result;

    REQUIRE("function_not_null", f != NULL);
    REQUIRE("node_not_null", node != NULL);
    model = (bates_model *)f->args;
    y = node->value[Y_DIM];
    sigma = model->sigma;
    result = 0.5 * POW(sigma, 2.) * y;

    return result;
}

static double func_ux(const function *f, const grid_node *node)
{
    bates_model *model;

    double x;
    double r;
    double a;
    double m;
    double delta;
    double lambda;
    double result;

    REQUIRE("function_not_null", f != NULL);
    REQUIRE("node_not_null", node != NULL);
```

```

    model = (bates_model *)f->args;
    x = node->value[X_DIM];
    r = model->r;
    a = model->alpha;
    m = model->m;
    delta = model->delta;
    lambda = model->lambda;
    result = (r - delta - lambda * (exp(0.5 * POW(a, 2.0) + m) - 1.0)) * x * (1.0

    return result;
}

static double func_uy(const function *f, const grid_node *node)
{

    bates_model *model = (bates_model *)f->args;

    double x = node->value[X_DIM];
    double y = node->value[Y_DIM];
    double Ks = model->Ks;
    double theta = model->theta;
    double rho = model->rho;
    double sigma = model->sigma;

    double result = rho * sigma * y * x + Ks * (theta - y);

    REQUIRE("function_not_null", f != NULL);
    REQUIRE("node_not_null", node != NULL);
    return result;
}

static double func_u(const function *f, const grid_node *node)
{

    bates_model *model = (bates_model *)f->args;

    double x = node->value[X_DIM];
    double r = model->r;
    double delta = model->delta;
    double a = model->alpha;

```

```

double m = model->m;
double lambda = model->lambda;

double result = (r - delta - lambda * (exp(0.5 * POW(a, 2.0) + m) - 1.0)) * x
  REQUIRE("function_not_null", f != NULL);
  REQUIRE("node_not_null", node != NULL);

  return result;
}

/* public interface */

int FDMertonHeston(double St0, NumFunc_1 *p, double T, double r, double divid,
{
  bates_model model;

  double K = p->Par[0].Val.V_DOUBLE;

  grid *grid;
  grid_tuner *tuner;
  boundary_description *boundary;
  pde_problem *problem;
  pde *equation;
  pde_term *pterm;
  pde_integral_term *iterm;
  stencil_operator *stnop;
  function *f_uxx, *f_uxy, *f_uyy, *f_ux, *f_uy, *f_u;
  function *f_payoff, *f_boundary;

  /*****
  *      MODEL      *
  *****/

  model.T = T;
  model.rho = rho;
  model.sigma = sigmav;
  model.theta = theta;
  model.r = r;
  model.K = K;
  model.Ks = kappa;
  model.delta = divid;

```

```

model.S0 = St0;
model.V0 = V0;
model.alpha = sqrt(v); /* CHECK THIS !!! */
model.m = m0;
model.lambda = lambda;
model.Ns = N1;
model.Nv = N2;

/*****
*      GRID and TUNER      *
*****/
grid_tuner_create(&tuner);
grid_tuner_set_argument(tuner, &model);
grid_tuner_set_tuner(tuner, EXPLICIT_TUNER, explicit_tuner_proc);
grid_tuner_set_tuner(tuner, IMPLICIT_TUNER, implicit_tuner_proc);
grid_tuner_set_tuner(tuner, RESCALE_TUNER, focus_rescaler_proc);

grid_create(&grid);
grid_set_space_dimensions(grid, 2);
grid_set_tuner(grid, tuner);
grid_set_min_value(grid, T_DIM, 0.0);
grid_set_max_value(grid, T_DIM, model.T);
grid_set_ticks(grid, X_DIM, model.Ns);
grid_set_ticks(grid, Y_DIM, model.Nv);
grid_set_iterator(grid, X_DIM, ITER_PLAIN);
grid_set_iterator(grid, Y_DIM, ITER_CORE);

/* focus */
grid_set_focus(grid, X_DIM, model.S0);
grid_set_focus(grid, Y_DIM, model.V0);
grid_rescale(grid);

/*****
*      BOUNDARY      *
*****/

cps_function_create(&f_payoff);
cps_function_set_args(f_payoff, &model);
cps_function_create(&f_boundary);
cps_function_set_args(f_boundary, &model);

```

```

cps_function_set_body(f_payoff, func_call_payoff);
cps_function_set_body(f_boundary, func_call_boundary);

boundary_description_create(&boundary);
boundary_description_set_left(boundary, X_DIM, f_boundary);
boundary_description_set_left(boundary, Y_DIM, f_boundary);
boundary_description_set_right(boundary, X_DIM, f_boundary);
boundary_description_set_right(boundary, Y_DIM, f_boundary);
boundary_description_set_initial(boundary, f_payoff);

/*****
*      EQUATION      *
*****/
pde_create(&equation);

/* 1: Uxx */

cps_function_create(&f_uxx);
cps_function_set_body(f_uxx, func_uxx);
stencil_operator_create(&stnop, STENCIL_OP_UXX);

pde_term_create(&pterm, UXX_TERM, f_uxx, stnop);
pde_add_term(equation, pterm);

/* 2: Uxy */
cps_function_create(&f_uxy);
cps_function_set_args(f_uxy, &model);
cps_function_set_body(f_uxy, func_uxy);
stencil_operator_create(&stnop, STENCIL_OP_UXY);

pde_term_create(&pterm, UXY_TERM, f_uxy, stnop);
pde_add_term(equation, pterm);

/* 3: Uyy */
cps_function_create(&f_uyy);
cps_function_set_args(f_uyy, &model);
cps_function_set_body(f_uyy, func_uyy);
stencil_operator_create(&stnop, STENCIL_OP_UYY);

pde_term_create(&pterm, UYY_TERM, f_uyy, stnop);

```

```
pde_add_term(equation, pterm);

/* 4: Ux */
cps_function_create(&f_ux);
cps_function_set_args(f_ux, &model);
cps_function_set_body(f_ux, func_ux);
stencil_operator_create(&stnop, STENCIL_OP_UX);

pde_term_create(&pterm, UX_TERM, f_ux, stnop);
pde_add_term(equation, pterm);

/* 5: Uy */
cps_function_create(&f_uy);
cps_function_set_args(f_uy, &model);
cps_function_set_body(f_uy, func_uy);
stencil_operator_create(&stnop, STENCIL_OP_UY);

pde_term_create(&pterm, UY_TERM, f_uy, stnop);
pde_add_term(equation, pterm);

/* 6: U */
cps_function_create(&f_u);
cps_function_set_args(f_u, &model);
cps_function_set_body(f_u, func_u);
stencil_operator_create(&stnop, STENCIL_OP_U);

pde_term_create(&pterm, U_TERM, f_u, stnop);
pde_add_term(equation, pterm);

/* 7: integral term */
if (model.lambda != 0.0)
{
    pde_integral_term_create(&iterm);
    pde_integral_term_set_lambda(iterm, model.lambda);
    pde_integral_term_set_alpha(iterm, model.alpha);
    pde_integral_term_set_m(iterm, model.m);
    pde_integral_term_set_grid(iterm, grid);

    pde_set_integral_term(equation, iterm);
}
```

```

/*****
*      PROBLEM      *
*****/
pde_problem_create(&problem);
problem->max_explicit_steps = 20;
pde_problem_set_desired_accuracy(problem, 10e-8);
pde_problem_set_equation(problem, equation);
pde_problem_set_grid(problem, grid);
pde_problem_set_boundary(problem, boundary);

/*****
*      SOLUTION     *
*****/
pde_problem_setup(problem);
pde_problem_solve(problem);
pde_problem_get_solution(problem, ptprice);
pde_problem_get_delta_x(problem, ptdelta);

if ((p->Compute) == &Call) /* CALL EVALUATION */
{
    (*ptprice) *= 2.0 * model.S0;
}
else /* PUT EVALUATION */
{
    (*ptprice) *= (2.0 * model.S0);
    (*ptprice) += model.K * exp(-model.r) - model.S0;
}

/*****
*      CLEANUP      *
*****/
pde_problem_destroy(&problem);

cps_function_destroy(&f_payoff);
cps_function_destroy(&f_boundary);
cps_function_destroy(&f_uxx);
cps_function_destroy(&f_uxy);
cps_function_destroy(&f_uyy);
cps_function_destroy(&f_ux);
cps_function_destroy(&f_uy);
cps_function_destroy(&f_u);

```

```

    return OK;
}

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

    if (ptMod->Sigma.Val.V_PDOUBLE == 0.0)
    {
        Fprintf(TOSCREEN, "BLACK-SCHOLES MODEL\ n\ n\ n");
        return WRONG;
    }
    else
    {
        r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
        divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

        return FDMertonHeston(ptMod->S0.Val.V_PDOUBLE,
                               ptOpt->PayOff.Val.V_NUMFUNC_1,
                               ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                               r,
                               divid, ptMod->Sigma0.Val.V_PDOUBLE
                               , ptMod->MeanReversion.Val.V_PDOUBLE,
                               ptMod->LongRunVariance.Val.V_PDOUBLE,
                               ptMod->Sigma.Val.V_PDOUBLE,
                               ptMod->Rho.Val.V_PDOUBLE,
                               ptMod->Lambda.Val.V_PDOUBLE,
                               ptMod->Mean.Val.V_PDOUBLE,
                               ptMod->Variance.Val.V_PDOUBLE,
                               Met->Par[0].Val.V_INT, Met->Par[1].Val.V_INT,
                               &(Met->Res[0].Val.V_DOUBLE),
                               &(Met->Res[1].Val.V_DOUBLE));
    }
}

static int CHK_OPT(FD_MertonHeston)(void *Opt, void *Mod)

```

```

{
    if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt
        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 = 51;
        Met->Par[1].Val.V_INT2 = 21;
    }
    return OK;
}
PricingMethod MET(FD_MertonHeston) =
{
    "FD_NataliniBriani_MERHES",
    {"SpaceStepNumber S", INT2, {100}, ALLOW}, {"SpaceStepNumber V", INT2, {100},
    CALC(FD_MertonHeston),
    { {"Price", DOUBLE, {100}, FORBID},
      {"Delta", DOUBLE, {100}, FORBID} ,
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(FD_MertonHeston),
    CHK_ok,
    MET(Init)
};

```