Help

```
// Written by P. Tankov and J. Poirot, June-September 2006
// This file is part of PREMIA software copying and usage restrictions apply

extern "C" {
#include "temperedstable1d_std.h"
#include "enums.h"
}
#include <cmath>
#include "math/cgmy/cgmy.h"
#include "math/cgmy/rnd.h"
#include "pnl/pnl_cdf.h"

extern "C" {

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

  // Pricing a european put option on a stock driven by Tempered Stable process
  // By Monte Carlo using the algorithm by Poirot and Tankov (2006)
  // Input parameters
  // T          : option maturity
  // S0         : initial stock price
  // r          : interest rate
  // q          : dividend yield
  // K          : strike
  // type       : use 1 for call, any other value for put
  // alphap, alphan, lambdap, lambdan, cp, cn : process parameters
  // Ntraj      : number of Monte Carlo simulations
  // Output values
  // price, delta, and the standard deviations of MC estimates
  // return value: zero if success, nonzero if error
```

```
// 1 is returned if alphap or alphan is equal to 1 (this case is not supported
static int MonteCarlo_TankovPoirot(double S0, NumFunc_1  *p, double T, double
{
  double K;
  int type;
  double price, delta, stdprice, stddelta;
  int simulation_dim = 1;
  int init_mc;
  double alpha, z_alpha;

  if ((alphap == 1.) || (alphan == 1.)) return BAD_ALPHA_TEMPSTABLE;
  K = p->Par[0].Val.V_DOUBLE;
  if ((p->Compute) == &Put)
    type = 0;
  else
    type = 1;

  /* Value to construct the confidence interval */
  alpha = (1. - confidence) / 2.;
  z_alpha = pnl_inv_cdfnor(1. - alpha);

  /*MC sampling*/
  init_mc = pnl_rand_init(generator, simulation_dim, Ntraj);
  if (init_mc == OK)
    {

      price = 0;
      stdprice = 0;
      delta = 0;
      stddelta = 0;

      double gcp = -pnl_tgamma(2. - alphap) / alphap / (alphap - 1) * pow(lamb
      double gcn = -pnl_tgamma(2. - alphan) / alphan / (alphan - 1) * pow(lamb
      double c = -pnl_tgamma(2. - alphap) / alphap / (alphap - 1) * pow(lambda
      double sigmap = pow(-cp * T * pnl_tgamma(2. - alphap) / alphap / (alphap
      double sigman = pow(-cn * T * pnl_tgamma(2. - alphan) / alphan / (alphan
      double mup = gcp * T - cp * T * pnl_tgamma(2. - alphap) / (1. - alphap)
      double mun = gcn * T + cn * T * pnl_tgamma(2. - alphan) / (1. - alphan)
      /*double stdconst = exp(pnl_tgamma(2.-alphap)/alphap/(alphap-1)*pow(lamb
      double XTP, XTN, XT, WT;
      /*double m = log(K/S0)-(r-divid)*T;
```

```
         double R;*/
        StableRnd Pos(alphap, sigmap, 1, mup, generator);
        StableRnd Neg(alphan, sigman, -1, mun, generator);
        for (long i = 0; i < Ntraj; i++)
          {
            XTP = Pos.next();
            XTN = Neg.next();
            XT = XTP + XTN;
            WT = exp(-lambdap * XTP + lambdan * XTN - c * T);
            double payoff = (K * exp(-r * T) - S0 * exp(-divid * T + XT)) * WT;
            if (payoff > 0)
              {
                price += (payoff / Ntraj);
                stdprice += (payoff * payoff / Ntraj);
                delta -= (exp(-divid * T + XT) * WT / Ntraj);
                stddelta += (exp(-2 * divid * T + 2 * XT) * WT * WT / Ntraj);
              }
          }
        stdprice = sqrt((1. / (Ntraj - 1)) * (stdprice - price * price));
        stddelta = sqrt((1. / (Ntraj - 1)) * (stddelta - delta * delta));
        if (type == 1)
          {
            price += S0 * exp(-divid * T) - K * exp(-r * T);
            delta += exp(-divid * T);
          }

        *ptprice = price;
        *ptdelta = delta;


        /* Price Confidence Interval */
        *inf_price = *ptprice - z_alpha * (stdprice);
        *sup_price = *ptprice + z_alpha * (stdprice);

        /* Delta Confidence Interval */
        *inf_delta = *ptdelta - z_alpha * (stddelta);
        *sup_delta = *ptdelta + z_alpha * (stddelta);
      }
    return OK;

  }
```

```
  int CALC(MC_TankovPoirot)(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  MonteCarlo_TankovPoirot(ptMod->S0.Val.V_PDOUBLE, ptOpt->PayOff.Val.V
  }

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

    return WRONG;
  }

#endif //PremiaCurrentVersion
  static int MET(Init)(PricingMethod *Met, Option *Opt)
  {
    static int first = 1;

    if (first)
      {
        Met->Par[0].Val.V_LONG = 10000000;
        Met->Par[1].Val.V_ENUM.value = 0;
        Met->Par[1].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[2].Val.V_PDOUBLE = 0.95;
        first = 0;
      }
    return OK;
  }


  PricingMethod MET(MC_TankovPoirot) =
  {
    "MC_TankovPoirot",
```

```
    { {"N iterations", LONG, {100}, ALLOW},
      {"RandomGenerator (Quasi Random not allowed)", ENUM, {100}, ALLOW},
      {"Confidence Value", DOUBLE, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CALC(MC_TankovPoirot),
    { {"Price", DOUBLE, {100}, FORBID},
      {"Delta", DOUBLE, {100}, FORBID},
      {"Inf Price", DOUBLE, {100}, FORBID},
      {"Sup Price", DOUBLE, {100}, FORBID} ,
      {"Inf Delta", DOUBLE, {100}, FORBID},
      {"Sup Delta", DOUBLE, {100}, FORBID},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },
    CHK_OPT(MC_TankovPoirot),
    CHK_mc,
    MET(Init)
  } ;
}
```