Help

```
extern "C" {
#include  "
href../../../mod/roughbergomi2d/roughbergomi2d_std/roughbergomi2d_std_h_src.pdfr
#include "
href../../../common/enums_h_src.pdfenums.h"
}
#include "pnl/pnl_complex.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_finance.h"
#include "pnl/pnl_fft.h"
#include <
href../../../common/math/highdim_solver/highdim_vector_h_src.pdfvector>
#include <
href../../../common/math/numerics_h_src.pdfnumeric>

/* The model considered is defined as
*
*  dS_t = \ sqrt{v_t} S_t dZ_t,
*  v_t = \ xi_0(t) \ exp\ left( \ eta \ tilde{W}_t - \ frac{1}{2} \ eta^2 t^{2H}
*
* where $Z, W$ are correlated Brownian motions with correlation \ rho.
* The process $\ tilde W$ is a variant of the fractional Brownian motion
* \ tilde{W}_t = \ int_0^t K(t,s) dW_s, \ quad K(t,s) = \ sqrt{2H} (t-s)^{H - 1/
*/

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2018+2) //The "#els
extern "C" {
static int CHK_OPT(MC_Bayer_RoughBergomi)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_Bayer_RoughBergomi)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
}
#else
```

```
typedef std::vector<double> Vector;

// Sample (Wtilde, W1). Here, Wtilde is a fBm, and W corresponds to a correlated
// More precisely, we have Wtilde[i] = \ int_0^{t_{i+1}} K(t_{i+1},s) dW_s for t
// On the other hand, W1[i] = (W_{t_{i+1}} - W_{t_i}) * \ sqrt{N}, i.e., scaled
// standard normal.
// This function uses the hybrid scheme.
// void sample_WtildeW_hybrid(Vector& Wtilde, Vector& W1, double H, Vector& Gamm
//  int N, RNorm& rnorm, int nDFT, fftw_complex* xC, fftw_complex* xHat, fftw_co
//  fftw_complex* yHat, fftw_complex* zC, fftw_complex* zHat,
//  fftw_plan& fPlanX, fftw_plan& fPlanY, fftw_plan& fPlanZ);

// Use the hybrid scheme to generate Wtilde from standard Gaussian samples W1, W
// Note that Wtilde here corresponds to samples of the (fBm) process Wtilde on a
// uniform grid of size N on [0,1], whereas W1, W1perp are i.i.d. N(0,1) gaussia
// Hence, both require rescaling.
static void compute_Wtilde(Vector& Wtilde, const Vector& W1, const Vector& W1per
    double H, Vector& Gamma, int N, int nDFT, PnlVect* xC, PnlVectComplex* xHat,
    PnlVect* yC, PnlVectComplex* yHat, PnlVect* zC,
    PnlVectComplex* zHat);

// Compute the Vector Gamma needed for the hybrid scheme
static Vector getGamma(int N, double H);

// Compute value on one single trajectory
static double updatePayoff(Vector& Wtilde, const Vector& W1, const Vector& W1per
    Vector& v, Vector& Gamma, PnlVect* xC, PnlVectComplex* xHat,
    PnlVect* yC, PnlVectComplex* yHat, PnlVect* zC,
    PnlVectComplex* zHat,  double eta, double H, double rho, double xi,
    double T, double K, int N);

// compute v
static void compute_V(Vector& v, const Vector& Wtilde, double H, double eta, dou
    double dt);

// compute \ int_0^T v_t dt
static double intVdt(const Vector & v, double dt);

// compute \ int_0^T \ sqrt{v_t} dW_t, with W = W1
static double intRootVdW(const Vector & v, const Vector & W1, double sdt);
```

```
// Auxiliary functions

// Copy real Vector to PnlVect array. xc is of size nDFT, x is of size <= nDFT
static void copyToPnlVect(const Vector& x, PnlVect* xc, size_t nDFT);

// Copy real part of fftw_complex array to Vector
static void copyFromPnlVect(Vector& x, const PnlVect* xc);



// Auxiliary functions

// z = a*x+b*y
template<typename T>
std::vector<T> linearComb(T a, const std::vector<T>& x, T b, const std::vector<T
  std::vector<T> z(x.size());
  for (size_t i = 0; i < z.size(); ++i)
    z[i] = a * x[i] + b * y[i];
  return z;
}

// x = s*x
template<typename T>
void scaleVector(std::vector<T>& x, T s) {
  for (size_t i = 0; i < x.size(); ++i)
    x[i] *= s;
}

static void rnorm(Vector &v, PnlRng *rng)
{
  for (size_t i = 0; i < v.size(); i++) {
    v[i] = pnl_rng_normal(rng);
  }
}

/*
 * Re-interpreting K as moneyness, we may have S0 = 1, without loss of generalit
 */
static void mc_bayer_roughbergomi_moneyness(double eta, double H, double rho,
    double xi, double K, double T, int M, int N, PnlRng *rng, double &price,
    double &stdev) {
```

```
  // Prepare random number generator

  // Prepare FFT-Plans
  int nDFT = 2 * N - 1; // size of arrays for DFT.
  // generate arrays of complex numbers for DFT
  PnlVect* xC = pnl_vect_create(nDFT);
  PnlVectComplex* xHat = pnl_vect_complex_create(nDFT);
  PnlVect* yC = pnl_vect_create(nDFT);
  PnlVectComplex* yHat = pnl_vect_complex_create(nDFT);
  PnlVect* zC = pnl_vect_create(nDFT);
  PnlVectComplex* zHat = pnl_vect_complex_create(nDFT);

  // Compute Gamma
  Vector Gamma = getGamma(N, H);

  // Allocate memory for Gaussian random numbers and the random vector v (instan
  // Note that W1, W1perp correspond to UNNORMALIZED increments of Brownian moti
  // i.e., are i.i.d. standard normal.
  Vector W1(N);
  Vector W1perp(N);
  Vector Wtilde(N);
  Vector WtildeScaled(N); // Wtilde scaled according to time
  Vector v(N);

  double mean = 0.0; // will eventually be the mean
  double mu2 = 0.0; // will become second moment
  double var; // will eventually become variance (i.e., MC error).

  // The big loop which needs to be parallelized in future
  for (int m = 0; m < M; ++m) {
    // generate the fundamental Gaussians
    rnorm(W1, rng);
    rnorm(W1perp, rng); // Wperp is  not needed!

    double payoff = updatePayoff(Wtilde, W1, W1perp, v, Gamma, xC, xHat, yC,
        yHat, zC, zHat, eta, H, rho, xi, T, K, N);
    mean += payoff;
    mu2 += payoff * payoff;
  }

// compute mean and variance
```

```
  mean = mean / M;
  mu2 = mu2 / M;
  var = mu2 - mean * mean;

// price = mean, stat = sqrt(var) / sqrt(M)
  price = mean;
  stdev = sqrt(var / M);

// free arrays of complex numbers
  pnl_vect_free(&xC);
  pnl_vect_complex_free(&xHat);
  pnl_vect_free(&yC);
  pnl_vect_complex_free(&yHat);
  pnl_vect_free(&zC);
  pnl_vect_complex_free(&zHat);
}

static void mc_bayer_roughbergomi(double S0, double eta, double H, double rho,
    double xi, double K, double T, int M, int N, PnlRng *rng, double &price,
    double &stdev)
{
  mc_bayer_roughbergomi_moneyness(eta, H, rho, xi, K / S0, T, M, N, rng, price,
  price *= S0;
  stdev *= S0;
}


void compute_Wtilde(Vector& Wtilde, const Vector& W1, const Vector& W1perp,
    double H, Vector& Gamma, int N, int nDFT, PnlVect* xC,
    PnlVectComplex* xHat, PnlVect* yC, PnlVectComplex* yHat,
    PnlVect* zC, PnlVectComplex* zHat)
{
  double s2H = sqrt(2.0 * H);
  double rhoH = s2H / (H + 0.5);
  Vector W1hat = linearComb(rhoH / s2H, W1, sqrt(1.0 - rhoH * rhoH) / s2H, W1per
  Vector Y2(N); // see R code
// Convolve W1 and Gamma
// Copy W1 and Gamma to complex arrays
  copyToPnlVect(W1, xC, nDFT);
  copyToPnlVect(Gamma, yC, nDFT);
// DFT both
```

```
  pnl_real_fft(xC, xHat);
  pnl_real_fft(yC, yHat);
// multiply xHat and yHat and save in zHat
  pnl_vect_complex_clone(zHat, xHat);
  pnl_vect_complex_mult_vect_term(zHat, yHat);
// inverse DFT zHat
  pnl_real_ifft(zHat, zC);
  copyFromPnlVect(Y2, zC);
// Wtilde = (Y2 + W1hat) * sqrt(2*H) * dt^H ??
  Wtilde = linearComb(sqrt(2.0 * H) * pow(1.0 / N, H), Y2,
      sqrt(2.0 * H) * pow(1.0 / N, H), W1hat);
}

double updatePayoff(Vector& Wtilde, const Vector& W1, const Vector& W1perp,
    Vector& v, Vector& Gamma, PnlVect* xC, PnlVectComplex* xHat,
    PnlVect* yC, PnlVectComplex* yHat, PnlVect* zC,
    PnlVectComplex* zHat,  double eta, double H, double rho, double xi,
    double T, double K, int N)
{
  double dt = T / N;
  double sdt = sqrt(dt);
  int nDFT = 2 * N - 1;
  compute_Wtilde(Wtilde, W1, W1perp, H, Gamma, N, nDFT, xC, xHat, yC, yHat, zC,
  scaleVector(Wtilde, pow(T, H)); // scale Wtilde for time T
  compute_V(v, Wtilde, H, eta, xi, dt); // compute instantaneous variance v
// now compute \ int v_s ds, \ int \ sqrt{v_s} dW_s with W = W1
  double Ivdt = intVdt(v, dt);
  double IsvdW = intRootVdW(v, W1, sdt);
// now compute the payoff by inserting properly into the BS formula
  double BS_vol = sqrt((1.0 - rho * rho) * Ivdt);
  double BS_spot = exp(-0.5 * rho * rho * Ivdt + rho * IsvdW);
  return pnl_bs_call(BS_spot, K, 1.0, 0., 0., BS_vol);
}

// Note that Wtilde plays the role of the old WtildeScaled!
void compute_V(Vector& v, const Vector& Wtilde, double H, double eta, double xi,
{
  v[0] = xi;
  for (size_t i = 1; i < v.size(); ++i)
    v[i] = xi * exp( eta * Wtilde[i - 1] - 0.5 * eta * eta * pow((i - 1) * dt, 2
}
```

```
double intVdt(const Vector & v, double dt) {
  return dt * std::accumulate(v.begin(), v.end(), 0.0);
}

double intRootVdW(const Vector & v, const Vector & W1, double sdt) {
  double IsvdW = 0.0;
  for (size_t i = 0; i < v.size(); ++i)
    IsvdW += sqrt(v[i]) * sdt * W1[i];
  return IsvdW;
}

Vector getGamma(int N, double H) {
  Vector Gamma(N);
  double alpha = H - 0.5;
  Gamma[0] = 0.0;
  for (int i = 1; i < N; ++i)
    Gamma[i] = (pow(i + 1.0, alpha + 1.0) - pow(i, alpha + 1.0))
        / (alpha + 1.0);
  return Gamma;
}

void copyToPnlVect(const Vector& x, PnlVect* xc, size_t nDFT) {
  for (size_t i = 0; i < x.size(); ++i) {
    LET(xc, i) = x[i];
  }
// fill up with 0s
  for (size_t i = x.size(); i < nDFT; ++i) {
    LET(xc, i) = 0.;
  }
}

void copyFromPnlVect(Vector& x, const PnlVect* xc) {
  for (size_t i = 0; i < x.size(); ++i)
    x[i] = GET(xc, i);
}


extern "C" {
int MCBayer_RoughBergomi(NumFunc_1  *p, double S0, double eta, double H, double
{
```

```
  //------------------Declaration of variable

  int call_put;
  double K;
  double price, stdev;
  PnlRng *rng;

  if ((p->Compute) == &Call)
    call_put = 1;
  else
    call_put = 0;
  K = p->Par[0].Val.V_PDOUBLE;
  rng = pnl_rng_create(generator);
  pnl_rng_sseed(rng, 0);

  mc_bayer_roughbergomi(S0, eta, H, rho, xi, K, T, M, N, rng, price, stdev);

  *ptprice=price;

  //Put case
  if(call_put==0)
   *ptprice = *ptprice - S0 + K ;

  pnl_rng_free(&rng);

  return OK;
}

int CALC(MC_Bayer_RoughBergomi)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;

  return MCBayer_RoughBergomi(ptOpt->PayOff.Val.V_NUMFUNC_1,
                              ptMod->S0.Val.V_PDOUBLE,
                              ptMod->sigma.Val.V_PDOUBLE,
                              ptMod->H.Val.V_PDOUBLE,
                              ptMod->rho.Val.V_PDOUBLE,
                              ptMod->sigma0.Val.V_PDOUBLE,
                              ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                              Met->Par[0].Val.V_LONG,
```

```
                                Met->Par[1].Val.V_PINT,
                                Met->Par[2].Val.V_ENUM.value,
                                &(Met->Res[0].Val.V_DOUBLE));
}

static int CHK_OPT(MC_Bayer_RoughBergomi)(void *Opt, void *Mod)
{

  if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt
    return OK;

  return  WRONG;
}
}
#endif //PremiaCurrentVersion


extern "C" {
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->Par[0].Val.V_LONG = 50000;
      Met->Par[1].Val.V_PINT = 200;
      Met->Par[2].Val.V_ENUM.value = 0;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->HelpFilenameHint = "mc_bayer_roughbergomi";
    }
  return OK;
}


PricingMethod MET(MC_Bayer_RoughBergomi) =
{
  "MC_Bayer_RoughBergomi",
  {
    {"Nb MC iterations", LONG, {100}, ALLOW},
    {"Nb discretisation steps", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
```

```
  },
  CALC(MC_Bayer_RoughBergomi),
  {
    {"Price", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_Bayer_RoughBergomi),
  CHK_mc,
  MET(Init)
};
}
```