```
 * RBergomi.cpp

#include "RBergomi.h"

RBergomi::RBergomi() {
    N = 0;
    M = 0;
    nDFT = 0;
    numThreads = 0;
    xi = 0.0;
    H = Vector(0);
    eta = Vector(0);
    rho = Vector(0);
    gen = std::vector<MTGenerator>(0);
    dist = std::vector<normDist>(0);
    xC = new fftw_complex*[0];
    xHat = new fftw_complex*[0];
    yC = new fftw_complex*[0];
    yHat = new fftw_complex*[0];
    zC = new fftw_complex*[0];
    zHat = new fftw_complex*[0];
    fPlanX = std::vector<fftw_plan>(0);
    fPlanY = std::vector<fftw_plan>(0);
    fPlanZ = std::vector<fftw_plan>(0);
    T = Vector(0);
    K = Vector(0);
}

// seed is optional
RBergomi::RBergomi(double x, Vector HIn, Vector e, Vector r, Vector t, Vector k, int NIn, long MIn,
        int numThreadsIn, std::vector<uint64_t> seed = std::vector<uint64_t>(0)) {
    // Some safety tests/asserts?
    N = NIn;
    nDFT = 2*N-1;
    M = MIn;
    numThreads = numThreadsIn;
    xi = x;
    H = HIn;
    eta = e;
    rho = r;
    gen = std::vector<MTGenerator>(numThreads);
    setGen(seed);
    dist = std::vector<normDist>(numThreads, normDist(0.0, 1.0));
    xC = new fftw_complex*[numThreads];
    xHat = new fftw_complex*[numThreads];
    yC = new fftw_complex*[numThreads];
```

```
    yHat = new fftw_complex*[numThreads];
    zC = new fftw_complex*[numThreads];
    zHat = new fftw_complex*[numThreads];
    for(int i=0; i<numThreads; ++i){
        xC[i] = new fftw_complex[nDFT];
        xHat[i] = new fftw_complex[nDFT];
        yC[i] = new fftw_complex[nDFT];
        yHat[i] = new fftw_complex[nDFT];
        zC[i] = new fftw_complex[nDFT];
        zHat[i] = new fftw_complex[nDFT];
    }
    fPlanX = std::vector<fftw_plan>(numThreads);
    fPlanY = std::vector<fftw_plan>(numThreads);
    fPlanZ = std::vector<fftw_plan>(numThreads);
    for(int i=0; i<numThreads; ++i){
        fPlanX[i] = fftw_plan_dft_1d(nDFT, xC[i], xHat[i], FFTW_FORWARD, FFTW_ESTIMATE);
        fPlanY[i] = fftw_plan_dft_1d(nDFT, yC[i], yHat[i], FFTW_FORWARD, FFTW_ESTIMATE);
        fPlanZ[i] = fftw_plan_dft_1d(nDFT, zHat[i], zC[i], FFTW_BACKWARD, FFTW_ESTIMATE);
    }
    T = t;
    K = k;

}

// delete allocated arrays
RBergomi::~RBergomi() {
    if(numThreads > 0){
        for(int i=0; i<numThreads; ++i){
            delete[] xC[i];
            delete[] xHat[i];
            delete[] yC[i];
            delete[] yHat[i];
            delete[] zC[i];
            delete[] zHat[i];
        }
    }
    delete[] xC;
    delete[] xHat;
    delete[] yC;
    delete[] yHat;
    delete[] zC;
    delete[] zHat;
}

Result RBergomi::ComputePriceST() {
```

```cpp
    // The random vectors; the first 3 are independent, Z is a composite
    // Note that W1, W1perp, Wperp, Z correspond to UNNORMALIZED increments of Brownian motions,
    // i.e., are i.i.d. standard normal.
    Vector W1(N);
    Vector W1perp(N);
    Vector Wperp(N);
    Vector Wtilde(N);
    Vector WtildeScaled(N); // Wtilde scaled according to time
    Vector ZScaled(N);
    // AT LEAST Z SHOULD BE SCALED AS WELL
    Vector v(N);
    Vector Z(N);
    double S; // maybe it is better to use a vector of S's corresponding to all different maturities!!
    // This would need a major re-organization of the code, including ParamTot...
    // The vectors of all combinations of parameter values and prices
    // bugfix: try to explicitly copy H before passing on
    Vector Hcopy = H;
    ParamTot par(Hcopy, eta, rho, T, K);
    // vectors of prices and variances
    Vector price(par.size(), 0.0);
    Vector var(par.size(), 0.0);
    // other parameters used
    double dt; // time increment
    double sdt; // square root of time increment

    // The big loop which needs to be parallelized in future
    for(int m = 0; m<M; ++m){
        // generate the fundamental Gaussians
        genGaussianST(W1);
        genGaussianST(W1perp);
        genGaussianST(Wperp);

        // now iterate through all parameters
        for(long i=0; i<par.size(); ++i){
            // Note that each of the changes here forces all subsequent updates!
            // check if H has changed. If so, Wtilde needs to be updated (and, hence, everything else)
            bool update = par.HTrigger(i);
            if(update)
                updateWtilde(Wtilde, W1, W1perp, par.H(i));
            // check if T has changed. If so, Wtilde and the time increment need re-scaling
            update = update || par.TTrigger(i);
            if(update){
```

```
                scaleWtilde(WtildeScaled, Wtilde, par.T(i), par.H(i));
                // THE OTHER RANDOM INCREMENTS SHOULD BE SCALED AS WELL!!!
                dt = par.T(i) / N;
                sdt = sqrt(dt);
            }
            // check if eta has changed. If so, v needs to be updated
            update = update || par.etaTrigger(i);
            if(update)
                updateV(v, WtildeScaled, par.H(i), par.eta(i), dt);
            // if rho has changed, then S needs to be re-computed
            update = update || par.rhoTrigger(i);
            if(update){
                updateZ(Z, W1, Wperp, par.rho(i));
                scaleZ(ZScaled, Z, sdt);
                S = updateS(v, ZScaled, dt);
            }
            // now compute the payoff
            //double payoff = par.K(i) > 1.0 ? posPart(par.K(i) - S) : posPart(S - par.K(i));
            double payoff = posPart(S - par.K(i)); // call option
            price[i] += payoff;
            var[i] += payoff*payoff;
        }
    }

    // compute mean and variance
    scaleVector(price, 1.0/double(M));
    scaleVector(var, 1.0/double(M)); // = E[X^2]
    var = linearComb(1.0, var, -1.0, squareVector(price)); // = empirical var of price
    Vector stat = rootVector(var);
    scaleVector(stat, 1.0/sqrt(double(M)));
    Vector iv(0);
    Result res{price, iv, par, stat, N, M, 1, 0.0};

    return res;
}

Result RBergomi::ComputeIVST() {
    return Result{};
}


Result RBergomi::ComputePrice() {
    return Result{};
}

Result RBergomi::ComputeIV() {
```

```
    return Result{};
}

void RBergomi::setGen(std::vector<uint64_t> seed) {
    if((seed.size() > 0) && (seed.size() != numThreads))
        std::cerr << "Wrong number of seeds provided! Switched back to default." << std::endl;
    // check whether seed was set
    if(seed.size() != numThreads){
        seed = std::vector<uint64_t>(numThreads);
        std::random_device r_dev;
        for(int i=0; i<numThreads; ++i)
            seed[i] = r_dev();
    }
    std::seed_seq seeder(seed.begin(), seed.end());
    seeder.generate(seed.begin(), seed.end());
    for(int i=0; i<numThreads; ++i){
        //seeder = std::seed_seq{seed[i]};
        gen[i].seed(seed[i]);
        //gen[i] = MTGenerator(seeder);
    }
}

void RBergomi::genGaussianST(Vector& X) {
    for(auto & x : X)
        x = dist[0](gen[0]); // replace 0 by omp_get_thread_num() for multi-threaded code
}

// Note that Wtilde[0] = 0!
void RBergomi::updateWtilde(Vector& Wtilde, const Vector& W1,
        const Vector& W1perp, double H) {
    Vector Gamma(N);
    getGamma(Gamma, H);
    double s2H = sqrt(2.0*H);
    double rhoH = s2H/(H+0.5);
    Vector W1hat = linearComb(rhoH/s2H, W1, sqrt(1.0 - rhoH*rhoH)/s2H, W1perp);
    Vector Y2(N); // see R code
    // Convolve W1 and Gamma
    // Copy W1 and Gamma to complex arrays
    copyToComplex(W1, xC[0]);
    copyToComplex(Gamma, yC[0]);
    // DFT both
    fftw_execute(fPlanX[0]); // DFT saved in xHat[0]
    fftw_execute(fPlanY[0]); // DFT saved in yHat[0]
    // multiply xHat and yHat and save in zHat
    complexMult(xHat[0], yHat[0], zHat[0]);
```

```
    // inverse DFT zHat
    fftw_execute(fPlanZ[0]);
    // read out the real part, re-scale by 1/nDFT
    copyToReal(Y2, zC[0]);
    scaleVector(Y2, 1.0/nDFT);
    // 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);
}

void RBergomi::scaleWtilde(Vector& WtildeScaled, const Vector& Wtilde,
        double T, double H) const {
    for(int i=0; i<N; ++i)
        WtildeScaled[i] = pow(T, H) * Wtilde[i];
}

void RBergomi::scaleZ(Vector& ZScaled, const Vector& Z, double sdt) const {
    for(int i=0; i<N; ++i)
        ZScaled[i] = sdt * Z[i];
}

void RBergomi::updateV(Vector& v, const Vector& WtildeScaled, double h,
        double e, double dt) const {
    for(int i=0; i<N; ++i)
        v[i] = xi * exp( e*WtildeScaled[i] - 0.5*e*e*pow(i*dt, 2*h));
}

void RBergomi::updateZ(Vector& Z, const Vector& W1, const Vector& Wperp,
        double r) const {
    Z = linearComb(r, W1, sqrt(1.0-r*r), Wperp);
}

double RBergomi::updateS(const Vector& v, const Vector& ZScaled, double dt) const {
    double X = 0.0;
    for(size_t i=0; i<v.size(); ++i)
        X += sqrt(v[i]) * ZScaled[i] - 0.5 * v[i] * dt;
    return exp(X); // Recall S_0 = 1.
}

void RBergomi::getGamma(Vector& Gamma, double H) const {
    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);
}
```

```
void RBergomi::copyToComplex(const Vector& x, fftw_complex* xc) {
    for(size_t i=0; i<x.size(); ++i){
        xc[i][0] = x[i]; // real part
        xc[i][1] = 0.0; // imaginary part
    }
    // fill up with 0es
    for(size_t i=x.size(); i<nDFT; ++i){
        xc[i][0] = 0.0; // real part
        xc[i][1] = 0.0; // imaginary part
    }
}

void RBergomi::copyToReal(Vector& x, const fftw_complex* xc) const {
    for(size_t i=0; i<x.size(); ++i)
         x[i] = xc[i][0]; // real part
}

void RBergomi::complexMult(const fftw_complex* x, const fftw_complex* y,
        fftw_complex* z) {
    for(size_t i=0; i<nDFT; ++i)
        fftw_c_mult(x[i], y[i], z[i]);
}

void RBergomi::fftw_c_mult(const fftw_complex a, const fftw_complex b,
        fftw_complex c){
    c[0] = a[0]*b[0] - a[1]*b[1];
    c[1] = a[0]*b[1] + a[1]*b[0];
}

void RBergomi::setXi(double xi) {
    this->xi = xi;
}

long RBergomi::getM() const {
    return M;
}

void RBergomi::setM(long m) {
    M = m;
}

int RBergomi::getN() const {
    return N;
}

void RBergomi::setN(int n) {
    N = n;
```

```
}

int RBergomi::getNumThreads() const {
    return numThreads;
}

void RBergomi::setNumThreads(int numThreads) {
    this->numThreads = numThreads;
}

double RBergomi::getXi() const {
    return xi;
}

void RBergomi::testScaleWtilde() {
    Vector W1(N);
    Vector Wtilde(N);
    Vector W1perp(N);
    Vector WtildeScaled(N); // Wtilde scaled according to time
    double H_scalar = H[0];
    double T_scalar = T[0];

    // names of files for W1, Wtilde, W1perp, WtildeScaled
    std::string f_W1 = "./W1.txt";
    std::string f_Wtilde = "./Wtilde.txt";
    std::string f_W1perp = "./W1perp.txt";
    std::string f_WtildeScaled = "./WtildeScaled.txt";
    std::fstream file;
    // make sure that files are empty
    file.open(f_W1.c_str(), std::fstream::out | std::fstream::trunc);
    file << "";
    file.close();
    file.open(f_Wtilde.c_str(), std::fstream::out | std::fstream::trunc);
    file << "";
    file.close();
    file.open(f_W1perp.c_str(), std::fstream::out | std::fstream::trunc);
    file << "";
    file.close();
    file.open(f_WtildeScaled.c_str(), std::fstream::out |
std::fstream::trunc);
    file << "";
    file.close();

    // The big loop which needs to be parallelized in future
    for(int m = 0; m<M; ++m){
        // generate the fundamental Gaussians
        genGaussianST(W1);
        genGaussianST(W1perp);
```

```
        updateWtilde(Wtilde, W1, W1perp, H_scalar);
        scaleWtilde(WtildeScaled, Wtilde, T_scalar, H_scalar);
        file.open(f_W1.c_str(), std::fstream::out | std::fstream::app);
        std::copy (W1.begin(), W1.end(), std::ostream_iterator<double>(file, " "));
        //file << "\n";
        file.close();
        file.open(f_Wtilde.c_str(), std::fstream::out | std::fstream::app);
        std::copy (Wtilde.begin(), Wtilde.end(), std::ostream_iterator<double>(file, " "));
        file << "\n";
        file.close();
        file.open(f_W1perp.c_str(), std::fstream::out | std::fstream::app);
        std::copy (W1perp.begin(), W1perp.end(), std::ostream_iterator<double>(file, " "));
        file << "\n";
        file.close();
        file.open(f_WtildeScaled.c_str(), std::fstream::out | std::fstream::app);
        std::copy (WtildeScaled.begin(), WtildeScaled.end(), std::ostream_iterator<double>(file, " "));
        file << "\n";
        file.close();
    }
}

void RBergomi::testConvolve() {
    // Convolve W1 and Gamma
    Vector Gamma(N);
    getGamma(Gamma, H[0]);
    // Choose special W1:
    Vector W1(N);
    for(int i=0; i<N; ++i)
        W1[i] = i;
    // Copy W1 and Gamma to complex arrays
    copyToComplex(W1, xC[0]);
    copyToComplex(Gamma, yC[0]);
    // DFT both
    fftw_execute(fPlanX[0]); // DFT saved in xHat[0]
    fftw_execute(fPlanY[0]); // DFT saved in yHat[0]
    // multiply xHat and yHat and save in zHat
    complexMult(xHat[0], yHat[0], zHat[0]);
    // inverse DFT zHat
    fftw_execute(fPlanZ[0]);
    // read out the real part, re-scale by 1/nDFT
    Vector Y2(N);
```

```
    copyToReal(Y2, zC[0]);
    scaleVector(Y2, 1.0/nDFT);
    std::cout << "Gamma = " << Gamma << std::endl;
    std::cout << "W1 = " << W1 << std::endl;
    std::cout << "Gamma * W1 = " << Y2 << std::endl;
}
```