

```

2  * RBergomi.cpp
7
8  #include "RBergomi.h"
9
10 RBergomi::RBergomi() {
11     N = 0;
12     M = 0;
13     nDFT = 0;
14     numThreads = 0;
15     xi = 0.0;
16     H = Vector(0);
17     eta = Vector(0);
18     rho = Vector(0);
19     gen = std::vector<MTGenerator>(0);
20     dist = std::vector<normDist>(0);
21     xC = new fftw_complex*[0];
22     xHat = new fftw_complex*[0];
23     yC = new fftw_complex*[0];
24     yHat = new fftw_complex*[0];
25     zC = new fftw_complex*[0];
26     zHat = new fftw_complex*[0];
27     fPlanX = std::vector<fftw_plan>(0);
28     fPlanY = std::vector<fftw_plan>(0);
29     fPlanZ = std::vector<fftw_plan>(0);
30     T = Vector(0);
31     K = Vector(0);
32 }
33
34 // seed is optional
35 RBergomi::RBergomi(double x, Vector HIn, Vector e, Vector r, Vector t,
36     Vector k, int NIn, long MIn,
37     int numThreadsIn, std::vector<uint64_t> seed =
38     std::vector<uint64_t>(0)) {
39     // Some safety tests/asserts?
40     N = NIn;
41     nDFT = 2*N-1;
42     M = MIn;
43     numThreads = numThreadsIn;
44     xi = x;
45     H = HIn;
46     eta = e;
47     rho = r;
48     gen = std::vector<MTGenerator>(numThreads);
49     setGen(seed);
50     dist = std::vector<normDist>(numThreads, normDist(0.0, 1.0));
51     xC = new fftw_complex*[numThreads];
52     xHat = new fftw_complex*[numThreads];
53     yC = new fftw_complex*[numThreads];
54     yHat = new fftw_complex*[numThreads];
55     zC = new fftw_complex*[numThreads];
56     zHat = new fftw_complex*[numThreads];
57     fPlanX = std::vector<fftw_plan>(numThreads);
58     fPlanY = std::vector<fftw_plan>(numThreads);
59     fPlanZ = std::vector<fftw_plan>(numThreads);
60     T = Vector(0);
61     K = Vector(0);
62 }

```

```

52     yHat = new fftw_complex*[numThreads];
53     zC = new fftw_complex*[numThreads];
54     zHat = new fftw_complex*[numThreads];
55     for(int i=0; i<numThreads; ++i){
56         xC[i] = new fftw_complex[nDFT];
57         xHat[i] = new fftw_complex[nDFT];
58         yC[i] = new fftw_complex[nDFT];
59         yHat[i] = new fftw_complex[nDFT];
60         zC[i] = new fftw_complex[nDFT];
61         zHat[i] = new fftw_complex[nDFT];
62     }
63     fPlanX = std::vector<fftw_plan>(numThreads);
64     fPlanY = std::vector<fftw_plan>(numThreads);
65     fPlanZ = std::vector<fftw_plan>(numThreads);
66     for(int i=0; i<numThreads; ++i){
67         fPlanX[i] = fftw_plan_dft_1d(nDFT, xC[i], xHat[i], FFTW_FORWARD,
FFTW_ESTIMATE);
68         fPlanY[i] = fftw_plan_dft_1d(nDFT, yC[i], yHat[i], FFTW_FORWARD,
FFTW_ESTIMATE);
69         fPlanZ[i] = fftw_plan_dft_1d(nDFT, zHat[i], zC[i], FFTW_BACKWARD,
FFTW_ESTIMATE);
70     }
71     T = t;
72     K = k;
73
74 }
75
76 // delete allocated arrays
77 RBergomi::~RBergomi() {
78     if(numThreads > 0){
79         for(int i=0; i<numThreads; ++i){
80             delete[] xC[i];
81             delete[] xHat[i];
82             delete[] yC[i];
83             delete[] yHat[i];
84             delete[] zC[i];
85             delete[] zHat[i];
86         }
87     }
88     delete[] xC;
89     delete[] xHat;
90     delete[] yC;
91     delete[] yHat;
92     delete[] zC;
93     delete[] zHat;
94 }
95
96 Result RBergomi::ComputePriceST() {

```

RBERGOMI.CPP

```

97 // The random vectors; the first 3 are independent, Z is a composite
98 // Note that W1, W1perp, Wperp, Z correspond to UNNORMALIZED
increments of Brownian motions,
99 // i.e., are i.i.d. standard normal.
100 Vector W1(N);
101 Vector W1perp(N);
102 Vector Wperp(N);
103 Vector Wtilde(N);
104 Vector WtildeScaled(N); // Wtilde scaled according to time
105 Vector ZScaled(N);
106 // AT LEAST Z SHOULD BE SCALED AS WELL
107 Vector v(N);
108 Vector Z(N);
109 double S; // maybe it is better to use a vector of S's corresponding
to all different maturities!!
110 // This would need a major re-organization of the code, including
ParamTot...
111 // The vectors of all combinations of parameter values and prices
112 // bugfix: try to explicitly copy H before passing on
113 Vector Hcopy = H;
114 ParamTot par(Hcopy, eta, rho, T, K);
115 // vectors of prices and variances
116 Vector price(par.size(), 0.0);
117 Vector var(par.size(), 0.0);
118 // other parameters used
119 double dt; // time increment
120 double sdt; // square root of time increment
121
122 // The big loop which needs to be parallelized in future
123 for(int m = 0; m<M; ++m){
124 // generate the fundamental Gaussians
125 genGaussianST(W1);
126 genGaussianST(W1perp);
127 genGaussianST(Wperp);
128
129 // now iterate through all parameters
130 for(long i=0; i<par.size(); ++i){
131 // Note that each of the changes here forces all subsequent
updates!
132 // check if H has changed. If so, Wtilde needs to be updated
(and, hence, everything else)
133 bool update = par.HTrigger(i);
134 if(update)
135 updateWtilde(Wtilde, W1, W1perp, par.H(i));
136 // check if T has changed. If so, Wtilde and the time
increment need re-scaling
137 update = update || par.TTrigger(i);
138 if(update){

```

RBERGOMI.CPP

```

139         scaleWtilde(WtildeScaled, Wtilde, par.T(i), par.H(i));
140         // THE OTHER RANDOM INCREMENTS SHOULD BE SCALED AS WELL!!!
141         dt = par.T(i) / N;
142         sdt = sqrt(dt);
143     }
144     // check if eta has changed. If so, v needs to be updated
145     update = update || par.etaTrigger(i);
146     if(update)
147         updateV(v, WtildeScaled, par.H(i), par.eta(i), dt);
148     // if rho has changed, then S needs to be re-computed
149     update = update || par.rhoTrigger(i);
150     if(update){
151         updateZ(Z, W1, Wperp, par.rho(i));
152         scaleZ(ZScaled, Z, sdt);
153         S = updateS(v, ZScaled, dt);
154     }
155     // now compute the payoff
156     //double payoff = par.K(i) > 1.0 ? posPart(par.K(i) - S) :
posPart(S - par.K(i));
157     double payoff = posPart(S - par.K(i)); // call option
158     price[i] += payoff;
159     var[i] += payoff*payoff;
160 }
161 }
162
163 // compute mean and variance
164 scaleVector(price, 1.0/double(M));
165 scaleVector(var, 1.0/double(M)); // = E[X^2]
166 var = linearComb(1.0, var, -1.0, squareVector(price)); // = empirical
var of price
167 Vector stat = rootVector(var);
168 scaleVector(stat, 1.0/sqrt(double(M)));
169 Vector iv(0);
170 Result res{price, iv, par, stat, N, M, 1, 0.0};
171
172 return res;
173 }
174
175 Result RBergomi::ComputeIVST() {
176     return Result{};
177 }
178
179
180 Result RBergomi::ComputePrice() {
181     return Result{};
182 }
183
184 Result RBergomi::ComputeIV() {

```

```

185     return Result{};
186 }
187
188 void RBergomi::setGen(std::vector<uint64_t> seed) {
189     if((seed.size() > 0) && (seed.size() != numThreads))
190         std::cerr << "Wrong number of seeds provided! Switched back to
default." << std::endl;
191     // check whether seed was set
192     if(seed.size() != numThreads){
193         seed = std::vector<uint64_t>(numThreads);
194         std::random_device r_dev;
195         for(int i=0; i<numThreads; ++i)
196             seed[i] = r_dev();
197     }
198     std::seed_seq seeder(seed.begin(), seed.end());
199     seeder.generate(seed.begin(), seed.end());
200     for(int i=0; i<numThreads; ++i){
201         //seeder = std::seed_seq{seed[i]};
202         gen[i].seed(seed[i]);
203         //gen[i] = MTGenerator(seeder);
204     }
205 }
206
207 void RBergomi::genGaussianST(Vector& X) {
208     for(auto & x : X)
209         x = dist[0](gen[0]); // replace 0 by omp_get_thread_num() for
multi-threaded code
210 }
211
212 // Note that Wtilde[0] = 0!
213 void RBergomi::updateWtilde(Vector& Wtilde, const Vector& W1,
214     const Vector& W1perp, double H) {
215     Vector Gamma(N);
216     getGamma(Gamma, H);
217     double s2H = sqrt(2.0*H);
218     double rhoH = s2H/(H+0.5);
219     Vector W1hat = linearComb(rhoH/s2H, W1, sqrt(1.0 - rhoH*rhoH)/s2H,
W1perp);
220     Vector Y2(N); // see R code
221     // Convolve W1 and Gamma
222     // Copy W1 and Gamma to complex arrays
223     copyToComplex(W1, xC[0]);
224     copyToComplex(Gamma, yC[0]);
225     // DFT both
226     fftw_execute(fPlanX[0]); // DFT saved in xHat[0]
227     fftw_execute(fPlanY[0]); // DFT saved in yHat[0]
228     // multiply xHat and yHat and save in zHat
229     complexMult(xHat[0], yHat[0], zHat[0]);

```

```

230 // inverse DFT zHat
231 fftw_execute(fPlanZ[0]);
232 // read out the real part, re-scale by 1/nDFT
233 copyToReal(Y2, zC[0]);
234 scaleVector(Y2, 1.0/nDFT);
235 // Wtilde = (Y2 + W1hat) * sqrt(2*H) * dt^H ??
236 Wtilde = linearComb(sqrt(2.0*H)*pow(1.0/N, H), Y2, sqrt(2.0*H)*pow
(1.0/N, H), W1hat);
237 }
238
239 void RBergomi::scaleWtilde(Vector& WtildeScaled, const Vector& Wtilde,
240     double T, double H) const {
241     for(int i=0; i<N; ++i)
242         WtildeScaled[i] = pow(T, H) * Wtilde[i];
243 }
244
245 void RBergomi::scaleZ(Vector& ZScaled, const Vector& Z, double sdt) const
246 {
247     for(int i=0; i<N; ++i)
248         ZScaled[i] = sdt * Z[i];
249 }
250
251 void RBergomi::updateV(Vector& v, const Vector& WtildeScaled, double h,
252     double e, double dt) const {
253     for(int i=0; i<N; ++i)
254         v[i] = xi * exp( e*WtildeScaled[i] - 0.5*e*e*pow(i*dt, 2*h));
255 }
256
257 void RBergomi::updateZ(Vector& Z, const Vector& W1, const Vector& Wperp,
258     double r) const {
259     Z = linearComb(r, W1, sqrt(1.0-r*r), Wperp);
260 }
261
262 double RBergomi::updateS(const Vector& v, const Vector& ZScaled, double
263     dt) const {
264     double X = 0.0;
265     for(size_t i=0; i<v.size(); ++i)
266         X += sqrt(v[i]) * ZScaled[i] - 0.5 * v[i] * dt;
267     return exp(X); // Recall S_0 = 1.
268 }
269
270 void RBergomi::getGamma(Vector& Gamma, double H) const {
271     double alpha = H - 0.5;
272     Gamma[0] = 0.0;
273     for(int i=1; i<N; ++i)
274         Gamma[i] = (pow(i+1.0, alpha + 1.0) - pow(i, alpha + 1.0))/(alpha
+ 1.0);
275 }

```

```

274
275 void RBergomi::copyToComplex(const Vector& x, fftw_complex* xc) {
276     for(size_t i=0; i<x.size(); ++i){
277         xc[i][0] = x[i]; // real part
278         xc[i][1] = 0.0; // imaginary part
279     }
280     // fill up with 0es
281     for(size_t i=x.size(); i<nDFT; ++i){
282         xc[i][0] = 0.0; // real part
283         xc[i][1] = 0.0; // imaginary part
284     }
285 }
286
287 void RBergomi::copyToReal(Vector& x, const fftw_complex* xc) const {
288     for(size_t i=0; i<x.size(); ++i)
289         x[i] = xc[i][0]; // real part
290 }
291
292 void RBergomi::complexMult(const fftw_complex* x, const fftw_complex* y,
293     fftw_complex* z) {
294     for(size_t i=0; i<nDFT; ++i)
295         fftw_c_mult(x[i], y[i], z[i]);
296 }
297
298 void RBergomi::fftw_c_mult(const fftw_complex a, const fftw_complex b,
299     fftw_complex c){
300     c[0] = a[0]*b[0] - a[1]*b[1];
301     c[1] = a[0]*b[1] + a[1]*b[0];
302 }
303
304 void RBergomi::setXi(double xi) {
305     this->xi = xi;
306 }
307
308 long RBergomi::getM() const {
309     return M;
310 }
311
312 void RBergomi::setM(long m) {
313     M = m;
314 }
315
316 int RBergomi::getN() const {
317     return N;
318 }
319
320 void RBergomi::setN(int n) {
321     N = n;

```

```

322 }
323
324 int RBergomi::getNumThreads() const {
325     return numThreads;
326 }
327
328 void RBergomi::setNumThreads(int numThreads) {
329     this->numThreads = numThreads;
330 }
331
332 double RBergomi::getXi() const {
333     return xi;
334 }
335
336 void RBergomi::testScaleWtilde() {
337     Vector W1(N);
338     Vector Wtilde(N);
339     Vector W1perp(N);
340     Vector WtildeScaled(N); // Wtilde scaled according to time
341     double H_scalar = H[0];
342     double T_scalar = T[0];
343
344     // names of files for W1, Wtilde, W1perp, WtildeScaled
345     std::string f_W1 = "./W1.txt";
346     std::string f_Wtilde = "./Wtilde.txt";
347     std::string f_W1perp = "./W1perp.txt";
348     std::string f_WtildeScaled = "./WtildeScaled.txt";
349     std::fstream file;
350     // make sure that files are empty
351     file.open(f_W1.c_str(), std::fstream::out | std::fstream::trunc);
352     file << "";
353     file.close();
354     file.open(f_Wtilde.c_str(), std::fstream::out | std::fstream::trunc);
355     file << "";
356     file.close();
357     file.open(f_W1perp.c_str(), std::fstream::out | std::fstream::trunc);
358     file << "";
359     file.close();
360     file.open(f_WtildeScaled.c_str(), std::fstream::out |
std::fstream::trunc);
361     file << "";
362     file.close();
363
364     // The big loop which needs to be parallelized in future
365     for(int m = 0; m<M; ++m){
366         // generate the fundamental Gaussians
367         genGaussianST(W1);
368         genGaussianST(W1perp);

```



```

369     updateWtilde(Wtilde, W1, W1perp, H_scalar);
370     scaleWtilde(WtildeScaled, Wtilde, T_scalar, H_scalar);
371     file.open(f_W1.c_str(), std::fstream::out | std::fstream::app);
372     std::copy(W1.begin(), W1.end(), std::ostream_iterator<double>
(file, " "));
373     //file << "\n";
374     file.close();
375     file.open(f_Wtilde.c_str(), std::fstream::out |
std::fstream::app);
376     std::copy(Wtilde.begin(), Wtilde.end(),
std::ostream_iterator<double>(file, " "));
377     file << "\n";
378     file.close();
379     file.open(f_W1perp.c_str(), std::fstream::out |
std::fstream::app);
380     std::copy(W1perp.begin(), W1perp.end(),
std::ostream_iterator<double>(file, " "));
381     file << "\n";
382     file.close();
383     file.open(f_WtildeScaled.c_str(), std::fstream::out |
std::fstream::app);
384     std::copy(WtildeScaled.begin(), WtildeScaled.end(),
std::ostream_iterator<double>(file, " "));
385     file << "\n";
386     file.close();
387 }
388 }
389
390 void RBergomi::testConvolve() {
391     // Convolve W1 and Gamma
392     Vector Gamma(N);
393     getGamma(Gamma, H[0]);
394     // Choose special W1:
395     Vector W1(N);
396     for(int i=0; i<N; ++i)
397         W1[i] = i;
398     // Copy W1 and Gamma to complex arrays
399     copyToComplex(W1, xC[0]);
400     copyToComplex(Gamma, yC[0]);
401     // DFT both
402     fftw_execute(fPlanX[0]); // DFT saved in xHat[0]
403     fftw_execute(fPlanY[0]); // DFT saved in yHat[0]
404     // multiply xHat and yHat and save in zHat
405     complexMult(xHat[0], yHat[0], zHat[0]);
406     // inverse DFT zHat
407     fftw_execute(fPlanZ[0]);
408     // read out the real part, re-scale by 1/nDFT
409     Vector Y2(N);

```

```
410     copyToReal(Y2, zC[0]);
411     scaleVector(Y2, 1.0/nDFT);
412     std::cout << "Gamma = " << Gamma << std::endl;
413     std::cout << "W1 = " << W1 << std::endl;
414     std::cout << "Gamma * W1 = " << Y2 << std::endl;
415 }
416
```