Help

```
#include <pnl/pnl_mathtools.h>
#include <pnl/pnl_complex.h>
#include"pnl/pnl_specfun.h"
#include "pnl/pnl_integration.h"
#include <pnl/pnl_vector.h>
#include <pnl/pnl_matrix.h>
#include <pnl/pnl_fft.h>
//#include <algorithm>
//#include "ap_barrier_proj.h"
#include "
href../../../common/math/kirkby/ap_asian_proj_h_src.pdfap_asian_proj.h"
#include "
href../../../common/math/kirkby/model_proj_h_src.pdfmodel_proj.h"
#include "
href../../../common/math/kirkby/proj_integrand_h_src.pdfproj_integrand.h"
#include <cmath>


////////////////////////////////////////////////////////////////////////////////
void fill_quadrature_matrix(PnlMatComplex* PSI, double dxi, int NNM, double left
for (int n = 0; n < NNM; n++) {
*pnl_mat_complex_lget(PSI, 0, n) = Complex(1.0, 0.0);
}

int Neta = 5 * NNM + 15;
int Neta5 = NNM + 3;
double g2 = sqrt(5. - 2. * sqrt(10. / 7.)) / 6.;
double g3 = sqrt(5. + 2. * sqrt(10. / 7.)) / 6.;
double v1 = .5 * 128. / 225;
double v2 = .5 * (322. + 13. * sqrt(70)) / 900.;
double v3 = .5 * (322. - 13. * sqrt(70)) / 900.;

// Initialize Quadrature Nodes
PnlVect* thet = pnl_vect_create_from_zero(Neta);

double cons1 = left_bound - 1.5 * dx;
for (int n = 0; n < Neta5; n++) {
double cons2 = cons1 + dx * n;
int index = 5 * n;
```

```
LET(thet, index) = cons2 - dx * g3;   // -4
LET(thet, index + 1) = cons2 - dx * g2; // -3

LET(thet, index + 2) = cons2;   // -2  (center)

LET(thet, index + 3) = cons2 + dx * g2; // -1
LET(thet, index + 4) = cons2 + dx * g3; // 0
}

// Initialize Quadrature Weights
PnlVect* sig = pnl_vect_create_from_list(10, -1.5 - g3, -1.5 - g2, -1.5,
-1.5 + g2, -1.5 + g3, -.5 - g3, -.5 - g2, -.5, -.5 + g2, -.5 + g3);

for (int n = 0; n < 5; n++) LET(sig, n) = pow((GET(sig, n) + 2), 3) / 6.0;
for (int n = 5; n < 10; n++) LET(sig, n) = 2.0 / 3 - 0.5 * pow(GET(sig, n), 3) -

LET(sig, 0) = v3 * GET(sig, 0); LET(sig, 4) = v3 * GET(sig, 4);
LET(sig, 5) = v3 * GET(sig, 5); LET(sig, 9) = v3 * GET(sig, 9);

LET(sig, 1) = v2 * GET(sig, 1); LET(sig, 3) = v2 * GET(sig, 3);
LET(sig, 6) = v2 * GET(sig, 6); LET(sig, 8) = v2 * GET(sig, 8);

LET(sig, 2) = v1 * GET(sig, 2); LET(sig, 7) = v1 * GET(sig, 7);

// Fill Quadrature Matrix
PnlVectComplex* zz = pnl_vect_complex_create_from_zero(Neta);
PnlVectComplex* thetC = pnl_vect_complex_create_from_zero(Neta);

for (int n = 0; n < Neta; n++) {
dcomplex z = Cexp(CRmul(CI, dxi * log(1 + exp(GET(thet, n)))));
LET_COMPLEX(zz, n) = z;
LET_COMPLEX(thetC, n) = z;
}

int index, l_index, r_index;
dcomplex sum, temp;

int N = PSI->m;
for (int j = 1; j < N - 1; j++) { // TODO: check if this should be j < N
for (int k = 0; k < NNM; k++) {
```

```
sum = CZERO;
index = 5 * k;  // base index of thetC vector

for (int p = 0; p < 10; p++) {
l_index = index + p;
r_index = index + 19 - p;
temp = RCmul(GET(sig, p), Cadd(GET_COMPLEX(thetC, l_index), GET_COMPLEX(thetC, r
sum = Cadd(sum, temp);
}

//*pnl_mat_complex_lget(PSI, j, k) = sum;
pnl_mat_complex_set(PSI, j, k, sum);
}
// Update thet
pnl_vect_complex_mult_vect_term(thetC, zz);  // thet = thet.*zz;
}

pnl_vect_free(&thet);
pnl_vect_free(&sig);

pnl_vect_complex_free(&thetC);
pnl_vect_complex_free(&zz);
}

void make_shifted_grid(PnlVect* x1, PnlVect* Nm, double ER, double M0, double dx
LET(x1, 0) = ER;  // initialize to the expected risk neutral drift

for (int m = 1; m < M0; m++) {
LET(x1, m) = ER + log(1 + exp(GET(x1, m - 1)));  // Uses Benhamou Shift

}

LET(Nm, 0) = 0;
for (int m = 0; m < M0; m++) {
int Nmm = (int)floor((GET(x1, m) - ER) / dx);
LET(x1, m) = ER + (1.0 - N / 2.) * dx + Nmm * dx;
LET(Nm, m) = Nmm;
}
}

void calc_beta(PnlVectComplex* grand, PnlVect* beta, const PnlVect* xi, const Pn
```

```
dcomplex ex, chf;

// Fill Fourier Integrand
LET_COMPLEX(grand, 0) = Complex(AA, 0.0);

for (int n = 1; n < N; n++) {
ex = Cexp(CRmul(CI, -leftBound * GET(xi, n)));
chf = GET_COMPLEX(PhiR, n);
LET_COMPLEX(grand, n) = RCmul(GET(zeta, n), Cmul(chf, ex));
}

// Calculate Beta (Projection Coefficients)
pnl_fft_inplace(grand);

for (int n = 0; n < N; n++) {
LET(beta, n) = pnl_vect_complex_get_real(grand, n);
}
}

void update_fourier_integrand(PnlVectComplex* grand, const PnlMatComplex* PSI, c
// This Step Calculates: PSI(:, left_index:left_index+N)*beta.*PhiR;

// TODO: Make this more efficient, complex library is slow (at least the way it
dcomplex sum;
for (int i = 0; i < N; i++) {
sum = CZERO;
for (int j = 0; j < N; j++) {
sum = Cadd(sum, CRmul(pnl_mat_complex_get(PSI, i, left_index + j), GET(beta, j))
}
LET_COMPLEX(grand, i) = Cmul(sum, GET_COMPLEX(PhiR, i));
}
}

////////////////////////////////////////////////////////////////////////////////

int price_asian_model(Model_proj* model, int call, double S_0, double W, int M0,
double T,  double* price, double* delta, int L1, int logN)
{
///////////////////////////////////
///// Algorithm parameters setup
///////////////////////////////////////
```

```
double dt = T / M0;
double ER = dt * model->get_rn_drift();  // Drift, used to center grid

//////////////////////////////////////
int N = (int)pow(2, logN);    // grid roughly centered on[c1 - alph, c1 + alph]

double r, q, alph, dx, a, A, C_aN;

r = model->get_interest_rate();
q = model->get_div_yield();
alph = model->get_truncation_alpha(L1, T);

dx = 2 * alph / (N - 1.0);   // Grid stepsize
a = 1 / dx;

A = 32.0 * pow(a, 4);
C_aN = A / N;

//////////////////////////////
// Set up Shifted Grid
//////////////////////////////
PnlVect* x1 = pnl_vect_create_from_zero(M0); ///// grid
PnlVect* Nm = pnl_vect_create_from_zero(M0); ///// grid shift index  (NOTE: coul

make_shifted_grid(x1, Nm, ER, M0, dx, N);
int NNM = N + (int)GET(Nm, M0 - 2);  // Total Number of columns of PSI

//////////////////////////////
// Intialize Characteristic Function of Log Return (PhiR)
//////////////////////////////
double dxi = 2 * M_PI * a / (double)N;
PnlVect* xi = pnl_vect_create_from_zero(N); ///// grid shift index

PnlVectComplex* PhiR = pnl_vect_complex_create_from_zero(N); ///// grid shift in
LET_COMPLEX(PhiR, 0) = Complex(1.0, 0.0);
dcomplex chf;

double xi_n = 0;
for (int n = 1; n < N ; n++) {
xi_n += dxi;
LET(xi, n) = xi_n;
```

```
model->rn_chf(dt, xi_n, &chf);
LET_COMPLEX(PhiR, n) = chf;
}

////////////////////////////
// Initialize Gaussian Quadrature Matrix (5-Point Gauss version)
////////////////////////////
int left_bound_index = 0;  // initialize index of current grid, based on grid sh
double AA = 1.0 / A;
double left_bound = GET(x1, 0);  // initialze xmin corresponding to grid shift i

PnlMatComplex* PSI = pnl_mat_complex_create_from_zero(N, NNM);
fill_quadrature_matrix(PSI, dxi, NNM, left_bound, dx);

////////////////////////////
// Intialize Projection Coefficients
////////////////////////////
PnlVectComplex* grand = pnl_vect_complex_create_from_zero(N);   // Fourier integ
PnlVect* zeta = pnl_vect_create_from_zero(N);  /// Represents the Dual Generator
PnlVect* beta = pnl_vect_create_from_zero(N);

// Make the zeta vector (ie the dual generator evaluated at frequency grid point
cubic_fourier_zeta(zeta, N, dxi, a);

// Initialize Beta Projection Coefficients
calc_beta(grand, beta, xi, zeta, PhiR, AA, N, left_bound);

// Multiply PhiR By C_aN, to incorporate the constant  (this is more efficient t
pnl_vect_complex_mult_double(PhiR, C_aN);
update_fourier_integrand(grand, PSI, beta, PhiR, left_bound_index, N);

////////////////////////////
// Main Loop
////////////////////////////
for (int m = 2; m < M0; m++) {
left_bound_index = (int)GET(Nm, m-1);  // Shift the x grid forward
left_bound = GET(x1, m - 1);  //  New leftmost x point on the shifted grid

calc_beta(grand, beta, xi, zeta, grand, AA, N, left_bound);  // update beta coef
update_fourier_integrand(grand, PSI, beta, PhiR, left_bound_index, N);  // updat
```

```
}

//////////////////////////////
// Final Price and Delta  (Price for S_0, and S_0 + epsilon to obtain a delta es
//////////////////////////////
// Payoff Constants
double c1, c2, c3, c4, e;
c1 = (exp(-7. / 4 * dx) / 54 + exp(-1.5 * dx) / 18 + exp(-1.25 * dx) / 2 + 7 * e

c2 = .05 * (28. / 27 + exp(-7. / 4 * dx) / 54 + exp(-1.5 * dx) / 18 + exp(-1.25
+ 121. / 54 * exp(-.75 * dx) + 23. / 18 * exp(-.5 * dx) + 235. / 54 * exp(-.25 *

c3 = ((28 + 7 * exp(-dx)) / 3 + (14 * exp(dx) + exp(-7. / 4 * dx) + 242 * cosh(.
+ .25 * (exp(-1.5 * dx) + 9 * exp(-1.25 * dx) + 46 * cosh(.5 * dx))) / 90;

c4 = (14. / 3 * (2 + cosh(dx))
+ .5 * (cosh(1.5 * dx) + 9 * cosh(1.25 * dx) + 23 * cosh(.5 * dx))
+ 1. / 6 * (cosh(7. / 4 * dx) + 121 * cosh(.75 * dx) + 235 * cosh(.25 * dx))) /

double eps = 1e-6;
double Svals[2] = { S_0, S_0 + eps };  // We price at two values of S_0, to obta
double prices[2] = {-100.0, -100.0};  // These will be set below

double S, ystar, C, D, val;

// Create a copy of the final integrand, so it doesnt get overwritten during del
PnlVectComplex* grand2 = pnl_vect_complex_create_from_zero(N);
for (int n = 0; n < N; n++) { LET_COMPLEX(grand2, n) = GET_COMPLEX(grand, n); };

for (int i = 0; i < 2; i++) {
S = Svals[i];
ystar = log((M0 + 1.0) * W / S - 1);
C = S / (M0 + 1.0);
D = W - C;

left_bound = GET(x1, M0-1);  // Get the last x1 shift
int nbar = (int)floor((ystar - left_bound)*a + 1) - 1;  // NOTE: -1 b/c grid is
left_bound = ystar - nbar * dx;

// Update Beta Coefficients (for final payoff evaluation, this is the final proj
calc_beta(grand, beta, xi, zeta, grand2, AA, N, left_bound);
```

```
// Calculate Price (of Put, and use Parity to price call)
val = 0.0;
for (int n = 0; n < nbar - 1; n++) {
e = exp(ystar + ((double)n - nbar) * dx);
val += GET(beta, n) * (D - C * c4 * e);
}

val += GET(beta, nbar-1) * ( D * 23. / 24 - C * c3 * exp(ystar - dx) );    // nb
val += GET(beta, nbar - 0) * (0.5 * D - C * c2 * exp(ystar));    // nbar - 0
val += GET(beta, nbar + 1) * (D / 24.0 - C * c1 * exp(ystar + dx));   // nbar +

val *= (C_aN * exp(-r * T));
if (call == 1) {  // Use Put-Call Parity to price a call
val += C * exp(-r * T) * (exp((r - q) * T * (1.0 + 1.0 / M0)) - 1) / (exp((r - q
}
prices[i] = val;
}

*price = prices[0];
*delta = (prices[1] - prices[0]) / eps;   // calc delta as finite difference

pnl_vect_free(&x1);
pnl_vect_free(&Nm);
pnl_vect_free(&xi);
pnl_vect_free(&zeta);
pnl_vect_free(&beta);

pnl_vect_complex_free(&PhiR);
pnl_vect_complex_free(&grand);
pnl_vect_complex_free(&grand2);

pnl_mat_complex_free(&PSI);

return OK;
}
int Kirkby_PROJ_kou_asian(int ifCall, double Spot, double sigma, double lambda,
double r, double divid,
double T, double Strike,
int logN, int M0, int L1,
double *ptprice, double *ptdelta)
```

```
{
Kou_Model_proj* model = new Kou_Model_proj(r, divid, sigma, lambda, P, lambdap,
int status = price_asian_model(model, ifCall, Spot, Strike, M0, T, ptprice, ptde
delete model;
return status;
}
int Kirkby_PROJ_NIG_asian(int ifCall, double Spot, double Sigma, double Theta, d
double r, double divid,
double T, double Strike,
int logN, int M0, int L1,
double *ptprice, double *ptdelta)
{
double delta = Sigma / sqrt(Kappa);
double beta = Theta / SQR(Sigma);
double alpha = sqrt(1 / (Kappa*SQR(Sigma)) + beta*beta);

NIG_Model_proj* model = new NIG_Model_proj(r, divid, alpha, beta, delta);
int status = price_asian_model(model, ifCall, Spot, Strike, M0, T, ptprice, ptde
delete model;
return status;
}
int Kirkby_PROJ_CGMY_asian(int ifCall, double Spot, double C, double G, double M
double r, double divid,
double T, double Strike,
int logN, int M0, int L1,
double *ptprice, double *ptdelta)
{
CGMY_Model_proj* model = new CGMY_Model_proj(r, divid, C, G, M, Y);
int status = price_asian_model(model, ifCall, Spot, Strike, M0, T, ptprice, ptde
delete model;
return status;
}
```