Help

```
#include<stdlib.h>
#include<math.h>
#include "pnl/pnl_random.h"
#include "pnl/pnl_specfun.h"
#include "variancegamma1d_pad.h"
#include "enums.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2011+2) //The "#els
static int CHK_OPT(MC_VarianceGamma_FloatingAsian)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_VarianceGamma_FloatingAsian)(void *Opt, void *Mod, PricingMethod *Me
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else
//Compute the positive or negative jump size between the smallest and the bigges
static double jump_generator_VG(double *cdf_jump_vect, double *cdf_jump_points,
{
  double z, v, y;
  int test, temp, l, j, q;
  test = 0;
  v = pnl_rand_uni(generator);
  y = cdf_jump_vect[cdf_jump_vect_size] * v;
  l = cdf_jump_vect_size / 2;
  j = cdf_jump_vect_size;
  z = 0;
  if (cdf_jump_vect[l] > y)
    {
      l = 0;
      j = cdf_jump_vect_size / 2;
    }
  if (v == 1)
    {
      z = cdf_jump_points[cdf_jump_vect_size];
    }
  if (v == 0)
    {
```

```
      z = cdf_jump_points[0];
    }
  if (v != 1 && v != 0)
    {
      while (test == 0)
        {
          if (cdf_jump_vect[l + 1] > y)
            {
              q = l;
              test = 1;
            }
          else
            {
              temp = (j - l - 1) / 2 + l;
              if (cdf_jump_vect[temp] > y)
                {
                  j = temp;
                  l = l + 1;
                }
              else
                {
                  l = temp * (temp > l) + (l + 1) * (temp <= l);
                }
            }
        }
      z = cdf_jump_points[q] * exp((y - cdf_jump_vect[q]) * exp(M_G * cdf_jump_p
    }
  return z;
}

static int VG_Mc_FloatingAsian(NumFunc_2 *P, double S0, double
                               T, double r, double divid, double sigma, double t
{
  double s, s1, eps, err, *Xg, *Xd, *jump_time_vect_p, *jump_time_vect_m, dpayof
  double cdf_jump_bound, drift, control, s2, s3, s4, s5, s6, u, u0, w1, w2, z, C
  double cov_payoff_control, var_payoff, var_control, cor_payoff_control, contro
  double *cdf_jump_points, *cdf_jump_vect_p, *cdf_jump_vect_m, min_M_G, var_dpay
  int i, j, jump_number, jump_number_p, jump_number_m, cdf_jump_vect_size, m1, m
  G = sqrt(2 / kappa + theta * theta / (sigma * sigma)) / sigma + theta / (sigma
  M = sqrt(2 / kappa + theta * theta / (sigma * sigma)) / sigma - theta / (sigma
  C = 1 / kappa;
```

```
  control_expec = exp((r - divid) * T) * S0;
  err = 1e-16;
  eps = 1e-1;
  cdf_jump_vect_size = 100000;
  s = 0;
  s1 = 0;
  s2 = 0;
  s3 = 0;
  s4 = 0;
  s5 = 0;
  s6 = 0;
  lambda_p = 0;
  lambda_m = 0;
  dpayoff = 0;
////////////////////////////////////////////////////////////////////////////
  lambda_p = C * pnl_sf_gamma_inc(0., eps * M); //positive jump intensity
  while (lambda_p * T < 15)
    {
      eps = eps * 0.9;
      lambda_p = C * pnl_sf_gamma_inc(0., eps * M);
    }
  lambda_m = C * pnl_sf_gamma_inc(0., eps * G); //negative jump intensity
  while (lambda_m * T < 15)
    {
      eps = eps * 0.9;
      lambda_m = C * pnl_sf_gamma_inc(0., eps * G);
    }
  lambda_p = C * pnl_sf_gamma_inc(0., eps * M);
  drift = (r - divid) + log(1 - (theta + sigma * sigma / 2) * kappa) / kappa + t
////////////////////////////////////////////////////////////////////////////
  cdf_jump_bound = 5;
  min_M_G = MIN(M, G);
  //Computation of the biggest jump that we tolerate
  while (C * exp(-min_M_G * cdf_jump_bound) / (min_M_G * cdf_jump_bound) > err)
    cdf_jump_bound++;
  pas = (cdf_jump_bound - eps) / cdf_jump_vect_size;
  cdf_jump_points = malloc((cdf_jump_vect_size + 1) * sizeof(double));
  cdf_jump_vect_p = malloc((cdf_jump_vect_size + 1) * sizeof(double));
  cdf_jump_vect_m = malloc((cdf_jump_vect_size + 1) * sizeof(double));
  cdf_jump_points[0] = eps;
  cdf_jump_vect_p[0] = 0;
```

```
  cdf_jump_vect_m[0] = 0;
  //computation of the cdf of the positive and negative jumps at some points
  for (i = 1; i <= cdf_jump_vect_size; i++)
    {
      cdf_jump_points[i] = i * pas + eps;
      cdf_jump_vect_p[i] = cdf_jump_vect_p[i - 1] + exp(-M * cdf_jump_points[i -
      cdf_jump_vect_m[i] = cdf_jump_vect_m[i - 1] + exp(-G * cdf_jump_points[i -
    }
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////
  m1 = (int)(1000 * lambda_p * T);
  m2 = (int)(1000 * lambda_m * T);
  jump_time_vect_p = malloc((m1) * sizeof(double));
  jump_time_vect_m = malloc((m2) * sizeof(double));
  jump_time_vect_p[0] = 0;
  jump_time_vect_m[0] = 0;
  Xg = malloc((m1 + m2) * sizeof(double)); //left value of X at jump times
  Xd = malloc((m1 + m2) * sizeof(double)); //right value of X at jump times
  Xg[0] = 0;
  Xd[0] = 0;
////////////////////////////////////////////////////
  pnl_rand_init(generator, 1, n_paths);
  /*Call Case*/
  if ((P->Compute) == &Call_StrikeSpot2)
    {
      for (i = 0; i < n_paths; i++)
        {
          //simulation of the positive jump times and number
          tau = -1 / (lambda_p) * log(pnl_rand_uni(generator));
          jump_number_p = 0;
          while (tau < T)
            {
              jump_number_p++;
              jump_time_vect_p[jump_number_p] = tau;
              tau += -1 / (lambda_p) * log(pnl_rand_uni(generator));
            }
          //simulation of the negative jump times and number
          tau = -1 / (lambda_m) * log(pnl_rand_uni(generator));
          jump_number_m = 0;
          while (tau < T)
            {
```

```
        jump_number_m++;
        jump_time_vect_m[jump_number_m] = tau;
        tau += -1 / (lambda_m) * log(pnl_rand_uni(generator));
      }
    jump_time_vect_p[jump_number_p + 1] = T;
    jump_time_vect_m[jump_number_m + 1] = T;
    jump_number = jump_number_p + jump_number_m;
//////////////////////////////////////////////////////////////
    //computation of Xg and Xd
    k1 = 1;
    k2 = 1;
    u0 = 0;
    for (k = 1; k <= jump_number; k++)
      {
        w1 = jump_time_vect_p[k1];
        w2 = jump_time_vect_m[k2];
        if (w1 < w2)
          {
            u = w1;
            k1++;
            z = jump_generator_VG(cdf_jump_vect_p, cdf_jump_points, cdf_ju
          }
        else
          {
            u = w2;
            k2++;
            z = -jump_generator_VG(cdf_jump_vect_m, cdf_jump_points, cdf_j
          }
        Xg[k] = drift * (u - u0) + Xd[k - 1];
        Xd[k] = Xg[k] + z;
        u0 = u;
      }
    Xg[jump_number + 1] = drift * (T - u0) + Xd[jump_number];
    Xd[jump_number + 1] = Xg[jump_number + 1];
/////////////////////////////////////////////////////////////
    //computation of the payoff
    payoff = 0;
    for (j = 1; j <= jump_number + 1; j++)
      {
        payoff += exp(Xg[j]) - exp(Xd[j - 1]);
      }
```

```
        dpayoff = exp(-r * T) * (exp(Xd[jump_number + 1]) - payoff / (drift *
        payoff = exp(-r * T) * S0 * (exp(Xd[jump_number + 1]) - payoff / (drif
        s1 += payoff;
        s += payoff * payoff;
        control = S0 * exp(Xd[jump_number + 1]);
        s2 += control;
        s3 += control * control;
        s4 += control * payoff;
        s5 += dpayoff;
        s6 += dpayoff * dpayoff;
      }
    cov_payoff_control = s4 / n_paths - s1 * s2 / ((double)n_paths * n_paths);
    var_payoff = (s - s1 * s1 / ((double)n_paths)) / (n_paths - 1);
    var_control = (s3 - s2 * s2 / ((double)n_paths)) / (n_paths - 1);
    cor_payoff_control = cov_payoff_control / (sqrt(var_payoff) * sqrt(var_con
    control_coef = cov_payoff_control / var_control;
    var_dpayoff = (s6 - s5 * s5 / ((double)n_paths)) / (n_paths - 1);
    *ptprice = (s1 / n_paths - control_coef * (s2 / n_paths - control_expec));
    *errorprice = 1.96 * sqrt(var_payoff * (1 - cor_payoff_control * cor_payof
    *ptdelta = s5 / (n_paths);
    *errordelta = 1.96 * sqrt(var_dpayoff) / sqrt(n_paths);
  }
/*Put case*/
if ((P->Compute) == &Put_StrikeSpot2)
  {
    for (i = 0; i < n_paths; i++)
      {
        //simulation of the positive jump times and number
        tau = -1 / (lambda_p) * log(pnl_rand_uni(generator));
        jump_number_p = 0;
        while (tau < T)
          {
            jump_number_p++;
            jump_time_vect_p[jump_number_p] = tau;
            tau += -1 / (lambda_p) * log(pnl_rand_uni(generator));
          }
        //simulation of the negative jump times and number
        tau = -1 / (lambda_m) * log(pnl_rand_uni(generator));
        jump_number_m = 0;
        while (tau < T)
          {
```

```
          jump_number_m++;
          jump_time_vect_m[jump_number_m] = tau;
          tau += -1 / (lambda_m) * log(pnl_rand_uni(generator));
        }
      jump_time_vect_p[jump_number_p + 1] = T;
      jump_time_vect_m[jump_number_m + 1] = T;
      jump_number = jump_number_p + jump_number_m; //total jump number
/////////////////////////////////////////////////////////////
      k1 = 1;
      k2 = 1;
      u0 = 0;
      //computation of Xg and Xd
      for (k = 1; k <= jump_number; k++)
        {
          w1 = jump_time_vect_p[k1];
          w2 = jump_time_vect_m[k2];
          if (w1 < w2)
            {
              u = w1;
              k1++;
              z = jump_generator_VG(cdf_jump_vect_p, cdf_jump_points, cdf_ju
            }
          else
            {
              u = w2;
              k2++;
              z = -jump_generator_VG(cdf_jump_vect_m, cdf_jump_points, cdf_j
            }
          Xg[k] = drift * (u - u0) + Xd[k - 1];
          Xd[k] = Xg[k] + z;
          u0 = u;
        }
      Xg[jump_number + 1] = drift * (T - u0) + Xd[jump_number];
      Xd[jump_number + 1] = Xg[jump_number + 1];
////////////////////////////////////////////////////////////
      //computation of the payoff
      payoff = 0;
      for (j = 1; j <= jump_number + 1; j++)
        {
          payoff += exp(Xg[j]) - exp(Xd[j - 1]);
        }
```

```
          dpayoff = exp(-r * T) * (payoff / (drift * T) - exp(Xd[jump_number + 1
          payoff = exp(-r * T) * S0 * (payoff / (drift * T) - exp(Xd[jump_number
          s1 += payoff;
          s += payoff * payoff;
          control = S0 * exp(Xd[jump_number + 1]);
          s2 += control;
          s3 += control * control;
          s4 += control * payoff;
          s5 += dpayoff;
          s6 += dpayoff * dpayoff;
        }
      cov_payoff_control = s4 / n_paths - s1 * s2 / ((double)n_paths * n_paths);
      var_payoff = (s - s1 * s1 / ((double)n_paths)) / (n_paths - 1);
      var_control = (s3 - s2 * s2 / ((double)n_paths)) / (n_paths - 1);
      cor_payoff_control = cov_payoff_control / (sqrt(var_payoff) * sqrt(var_con
      control_coef = cov_payoff_control / var_control;
      var_dpayoff = (s6 - s5 * s5 / ((double)n_paths)) / (n_paths - 1);
      *ptprice = (s1 / n_paths - control_coef * (s2 / n_paths - control_expec));
      *errorprice = 1.96 * sqrt(var_payoff * (1 - cor_payoff_control * cor_payof
      *ptdelta = s5 / (n_paths);
      *errordelta = 1.96 * sqrt(var_dpayoff) / sqrt(n_paths);
    }
  free(cdf_jump_vect_p);
  free(cdf_jump_vect_m);
  free(cdf_jump_points);
  free(jump_time_vect_p);
  free(jump_time_vect_m);
  free(Xd);
  free(Xg);
  return OK;
}

int CALC(MC_VarianceGamma_FloatingAsian)(void *Opt, void *Mod, PricingMethod *Me
{
  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  VG_Mc_FloatingAsian(ptOpt->PayOff.Val.V_NUMFUNC_2, ptMod->S0.Val.V_PDO
}

static int CHK_OPT(MC_VarianceGamma_FloatingAsian)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "AsianCallFloatingEuro") == 0) || (strcmp(((
    return OK;
  return WRONG;
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Mod)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->Par[0].Val.V_ENUM.value = 0;
      Met->Par[0].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[1].Val.V_LONG = 10000;
    }
  return OK;
}

PricingMethod MET(MC_VarianceGamma_FloatingAsian) =
{
  "MC_VG_AsianFloating",
  { {"RandomGenerator", ENUM, {100}, ALLOW},
    {"N iterations", LONG, {100}, ALLOW}, {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_VarianceGamma_FloatingAsian),
  {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID}, {"Price E
  CHK_OPT(MC_VarianceGamma_FloatingAsian),
  CHK_ok,
  MET(Init)
} ;
```