Help

```
#include<stdlib.h>
#include<math.h>
#include"pnl/pnl_random.h"
#include"pnl/pnl_specfun.h"
#include "cgmy1d_pad.h"
#include "enums.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2011+2) //The "#els
static int CHK_OPT(MC_CGMY_FixedAsian)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_CGMY_FixedAsian)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else
//Compute the positive or negative jump size between the smallest and the bigges
static double jump_generator_CGMY(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 = pow(1 / pow(cdf_jump_points[q], Y) - (y - cdf_jump_vect[q]) * Y * exp(
    }

  return z;
}

//(exp(x)-1)/x
static double p_func(double x)
{
  double s;
  int i, n;
  n = 1;
  s = 0;
  for (i = 0; i <= n; i++)
    s += pow(x, i) / pnl_fact(i + 1);
```

```
  return s;
}

//(4exp(x)+(2x-3)exp(2x)-1)/x^3
static double var_func(double x)
{
  double s;
  int i, n;
  n = 1;
  s = 0;
  for (i = 0; i <= n; i++)
    s += 4 * pow(x, i) / pnl_fact(i + 3) - 3 * pow(2., i + 3) * pow(x, i) / pnl_

  return s;
}

//exp(x)/x-(exp(x)-1)/x^2
static double cov_func(double x)
{
  double s;
  int i, n;
  n = 1;
  s = 0;
  for (i = 0; i <= n; i++)
    s += pow(x, i) * (1. / pnl_fact(i + 1) - 1. / pnl_fact(i + 2));

  return s;
}

static int CGMY_Mc_FixedAsian(NumFunc_2 *P, double S0, double T, double r, doubl
{
  double eps, s, s1, s2, s3, s4, s5, s6, payoff, dpayoff, control, discount, w1,
  double control_expec, lambda_m, cdf_jump_bound, pas, cov_payoff_control, var_p
  double cor_payoff_control, control_coef, var_dpayoff, *cdf_jump_points, *cdf_j
  double *cdf_jump_vect_m, *Xg, *Xd, tau, *jump_time_vect, *jump_time_vect_p, *j
  double var_temp, cov_temp, *vect_temp, g_temp, min_M_G, K;
  int i, j, k, jump_number_p, jump_number_m, jump_number, m1, m2, cdf_jump_vect_
  K = P->Par[0].Val.V_DOUBLE;
  discount = exp(-r * T);
  err = 1E-16;
  eps = 0.1;
```

```
  cdf_jump_vect_size = 100000;
  if (r - divid != 0)
    control_expec = S0 * (exp((r - divid) * T) - 1) / ((r - divid) * T);
  else
    control_expec = S0;
  jump_number = 0;
  s = 0;
  s1 = 0;
  s2 = 0;
  s3 = 0;
  s4 = 0;
  s5 = 0;
  s6 = 0;
  if (M < 1 || G < 1 || Y >= 2 || Y == 0)
    {
      printf("Function CGMY_Mc_FixedAsian: invalid parameters. We must have M>=1
    }
  lambda_p = C * pow(M, Y) * pnl_sf_gamma_inc(-Y, eps * M); //positive jump inte
  while (lambda_p * T < 10)
    {
      eps = eps * 0.9;
      lambda_p = C * pow(M, Y) * pnl_sf_gamma_inc(-Y, eps * M);
    }
  lambda_m = C * pow(G, Y) * pnl_sf_gamma_inc(-Y, eps * G); //negative jump inte
  while (lambda_m * T < 10)
    {
      eps = eps * 0.9;
      lambda_m = C * pow(G, Y) * pnl_sf_gamma_inc(-Y, eps * G);
    }
  lambda_p = C * pow(M, Y) * pnl_sf_gamma_inc(-Y, eps * M);
////////////////////////////////////////////////////////
  cdf_jump_bound = 1;
  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 * pow(cdf_jump_bound, 1 +
    cdf_jump_bound += cdf_jump_bound + 5;

  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 -
    }
////////////////////////////////////////////////////
  sigma = sqrt(C * (pow(M, Y - 2) * (pnl_sf_gamma(2 - Y) - pnl_sf_gamma_inc(2 -
  if (Y == 1)
    drift = (r - divid) - C * ((M - 1) * log(1. - 1 / M) + (G + 1) * log(1. + 1
  else
    drift = (r - divid) - C * pnl_sf_gamma(-Y) * (pow(M, Y) * (pow(1 - 1 / M, Y)
  drift = drift - C * (pow(M, Y - 1) * (pnl_sf_gamma_inc(1 - Y, eps * M) - pnl_s

////////////////////////////////////////////////////
  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;
  jump_time_vect = malloc((m1 + m2) * sizeof(double));
  vect_temp = malloc((m1 + m2) * sizeof(double));
  jump_time_vect[0] = 0;
  vect_temp[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_OverSpot2)
    {
      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;
          u = 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_CGMY(cdf_jump_vect_p, cdf_jump_points, cdf_
                }
              else
                {
                  u = w2;
                  k2++;
                  z = -jump_generator_CGMY(cdf_jump_vect_m, cdf_jump_points, cdf
```

```
              }
            g_temp = pnl_rand_normal(generator);
            if (fabs(drift * (u - u0)) < 1e-4)
              {
                var_temp = (u - u0) * (u - u0) * (u - u0) * var_func(drift * (
                cov_temp = (u - u0) * (u - u0) * cov_func(drift * (u - u0));
              }
            else
              {
                var_temp = (4 * exp(drift * (u - u0)) + (2 * drift * (u - u0)
                cov_temp = (u - u0) * exp(drift * (u - u0)) / drift - (exp(dri
              }
            jump_time_vect[k] = u;
            vect_temp[k] = cov_temp * g_temp / (sqrt(u - u0)) + sqrt(var_temp
            Xg[k] = drift * (u - u0) + sigma * g_temp * sqrt(u - u0) + Xd[k -
            Xd[k] = Xg[k] + z;
            u0 = u;
          }
        g_temp = pnl_rand_normal(generator);
        if (fabs(drift * (T - u0)) < 1e-4)
          {
            var_temp = (T - u0) * (T - u0) * (T - u0) * var_func(drift * (T -
            cov_temp = (T - u0) * (T - u0) * cov_func(drift * (T - u0));
          }
        else
          {
            var_temp = (4 * exp(drift * (T - u0)) + (2 * drift * (T - u0) - 3)
            cov_temp = (T - u0) * exp(drift * (T - u0)) / drift - (exp(drift *
          }
        jump_time_vect[jump_number + 1] = T;
        vect_temp[jump_number + 1] = cov_temp * g_temp / (sqrt(T - u0)) + sqrt
        Xg[jump_number + 1] = drift * (T - u0) + sigma * g_temp * sqrt(T - u0)
        Xd[jump_number + 1] = Xg[jump_number + 1];
////////////////////////////////////////////////////////
        //computation of the payoff
        payoff = 0;
        for (j = 1; j <= jump_number + 1; j++)
          {
            if (fabs(drift * (jump_time_vect[j] - jump_time_vect[j - 1])) < 1e
              payoff += exp(Xd[j - 1]) * (p_func(drift * (jump_time_vect[j] -
            else
```

```
            payoff += exp(Xd[j - 1]) * ((exp(drift * (jump_time_vect[j] - ju
          }
        control = S0 * payoff / T;
        dpayoff = -discount * (payoff / T) * (S0 * payoff / T < K);
        payoff = discount * (K - S0 * payoff / T) * (S0 * payoff / T < K);
        s1 += payoff;
        s += payoff * payoff;
        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);
    if (r != divid)
      *ptprice = (s1 / n_paths - control_coef * (s2 / n_paths - control_expec)
    else
      *ptprice = (s1 / n_paths - control_coef * (s2 / n_paths - control_expec)
    *priceerror = 1.96 * sqrt(var_payoff * (1 - cor_payoff_control * cor_payof
    if (r != divid)
      *ptdelta = s5 / (n_paths) + (exp(-divid * T) - exp(-r * T)) / ((r - divi
    else
      *ptdelta = s5 / (n_paths) + exp(-r * T);
    *deltaerror = 1.96 * sqrt(var_dpayoff) / sqrt(n_paths);
  }
/*Put case*/
if ((P->Compute) == &Put_OverSpot2)
  {
    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;
        u = 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_CGMY(cdf_jump_vect_p, cdf_jump_points, cdf_
              }
            else
              {
                u = w2;
                k2++;
                z = -jump_generator_CGMY(cdf_jump_vect_m, cdf_jump_points, cdf
              }
            g_temp = pnl_rand_normal(generator);
            if (fabs(drift * (u - u0)) < 1e-4)
              {
                var_temp = (u - u0) * (u - u0) * (u - u0) * var_func(drift * (
```

```
                cov_temp = (u - u0) * (u - u0) * cov_func(drift * (u - u0));
              }
            else
              {
                var_temp = (4 * exp(drift * (u - u0)) + (2 * drift * (u - u0)
                cov_temp = (u - u0) * exp(drift * (u - u0)) / drift - (exp(dri
              }
            jump_time_vect[k] = u;
            vect_temp[k] = cov_temp * g_temp / (sqrt(u - u0)) + sqrt(var_temp
            Xg[k] = drift * (u - u0) + sigma * g_temp * sqrt(u - u0) + Xd[k -
            Xd[k] = Xg[k] + z;
            u0 = u;
          }
        g_temp = pnl_rand_normal(generator);
        if (fabs(drift * (T - u0)) < 1e-4)
          {
            var_temp = (T - u0) * (T - u0) * (T - u0) * var_func(drift * (T -
            cov_temp = (T - u0) * (T - u0) * cov_func(drift * (T - u0));
          }
        else
          {
            var_temp = (4 * exp(drift * (T - u0)) + (2 * drift * (T - u0) - 3)
            cov_temp = (T - u0) * exp(drift * (T - u0)) / drift - (exp(drift *
          }
        jump_time_vect[jump_number + 1] = T;
        vect_temp[jump_number + 1] = cov_temp * g_temp / (sqrt(T - u0)) + sqrt
        Xg[jump_number + 1] = drift * (T - u0) + sigma * g_temp * sqrt(T - u0)
        Xd[jump_number + 1] = Xg[jump_number + 1];
////////////////////////////////////////////////////////////
        //computation of the payoff
        payoff = 0;
        for (j = 1; j <= jump_number + 1; j++)
          {
            if (fabs(drift * (jump_time_vect[j] - jump_time_vect[j - 1])) < 1e
              payoff += exp(Xd[j - 1]) * (p_func(drift * (jump_time_vect[j] -
            else
              payoff += exp(Xd[j - 1]) * ((exp(drift * (jump_time_vect[j] - ju
          }
        control = S0 * payoff / T;
        dpayoff = -discount * (payoff / T) * (S0 * payoff / T < K);
        payoff = discount * (K - S0 * payoff / T) * (S0 * payoff / T < K);
```

```
          s1 += payoff;
          s += payoff * payoff;
          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));
      *priceerror = 1.96 * sqrt(var_payoff * (1 - cor_payoff_control * cor_payof
      *ptdelta = s5 / (n_paths);
      *deltaerror = 1.96 * sqrt(var_dpayoff) / sqrt(n_paths);
    }
  free(Xd);
  free(Xg);
  free(cdf_jump_points);
  free(cdf_jump_vect_p);
  free(cdf_jump_vect_m);
  free(jump_time_vect_p);
  free(jump_time_vect_m);
  free(jump_time_vect);
  free(vect_temp);

  return OK;
}
int CALC(MC_CGMY_FixedAsian)(void *Opt, void *Mod, PricingMethod *Met)
{
  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  CGMY_Mc_FixedAsian(ptOpt->PayOff.Val.V_NUMFUNC_2, ptMod->S0.Val.V_PDOU
```

```
}

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

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Mod)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->HelpFilenameHint = "mc_cgmy_asianfixed";
      Met->Par[0].Val.V_ENUM.value = 0;
      Met->Par[0].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[1].Val.V_LONG = 100000;
    }
  return OK;
}

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