Help

```
#include "mer1d_pad.h"
#include "pnl/pnl_cdf.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2009+2) //The "#els
static int CHK_OPT(MC_Merton_FloatingLookback)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_Merton_FloatingLookback)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else
static int Merton_Mc_FloatingLookback(double s_maxmin, NumFunc_2 *P, double S0,
{
  double *s, s1, s2, *s3, **s4, s5, s6;
  double pas, sup, inf, supw, infw, exp_sup, exp_inf, exp_supw, exp_infw, beta;
  double *t, *jump_size_vect, *jump_time_vect, sup0, temp, nu, discount, *cov_pa
  double cor_payoff_control, *coef_control, var_proba, determinant_control;
  double *X, *W, payoff, *control, proba, s0, s_shift, e1, e2, e3, expect_contro
  int i, j, k, l, jump_number, *N;
  k = 0;
  beta = 0.5826;
  delta = sqrt(var);
  nu = (r - divid) - sigma * sigma / 2 - lambda * (exp(var / 2 - mean) - 1);
  discount = exp(-r * T);
  pas = T / n_points;
  X = malloc((n_points + 1) * sizeof(double));
  W = malloc((n_points + 1) * sizeof(double));
  t = malloc((n_points + 1) * sizeof(double));
  N = malloc((n_points + 1) * sizeof(int));
  s = malloc((2) * sizeof(double));
  s3 = malloc((2) * sizeof(double));
  control = malloc((2) * sizeof(double));
  var_control = (double **) malloc((2) * sizeof(double));
  s4 = (double **) malloc((2) * sizeof(double));
  for (i = 0; i <= 1; i++)
```

```
  {
    var_control[i] = malloc((2) * sizeof(double));
    s4[i] = malloc((2) * sizeof(double));
  }
cov_payoff_control = malloc((2) * sizeof(double));
coef_control = malloc((2) * sizeof(double));
jump_size_vect = malloc((int)(1000 * lambda * T) * sizeof(double));
jump_time_vect = malloc((int)(1000 * lambda * T) * sizeof(double));
N[0] = 0;
X[0] = 0;
s[0] = 0;
s[1] = 0;
s3[0] = 0;
s3[1] = 0;
s4[0][0] = 0;
s4[0][1] = 0;
s4[1][0] = 0;
s4[1][1] = 0;
s1 = 0;
s2 = 0;
s5 = 0;
s6 = 0;
for (k = 0; k <= n_points; k++)
  {
    t[k] = k * pas;
  }
pnl_rand_init(generator, 1, n_paths);
//Call options case
if ((P->Compute) == &Call_StrikeSpot2)
  {
    s_shift = exp(beta * sigma * sqrt(T / n_points)) * s_maxmin;
    e1 = (log(s_maxmin / S0) * (S0 > s_maxmin) - (nu + sigma * sigma) * T) / (
    e2 = e1 + sigma * sqrt(T);
    e3 = (log(s_maxmin / S0) * (S0 > s_maxmin) + nu * T) / (sigma * sqrt(T));
    //first control variable's expected value : it's the exponential of the in
    if (S0 >= s_maxmin)
      expect_control_0 = exp(beta * sigma * sqrt(T / n_points)) * (discount *
    else
      expect_control_0 = exp(beta * sigma * sqrt(T / n_points)) * (S0 * exp(si
    for (i = 1; i <= n_paths; i++)
      {
```

```
jump_number = pnl_rand_poisson(lambda * T, generator); // number of ju
W[0] = 0;
// simulation of the continuous part of the Levy process : Brownian mo
for (j = 1; j <= n_points; j++)
  {
    W[j] = sigma * pnl_rand_normal(generator) * sqrt(pas) + nu * pas +
  }
// simulation of the jump's times and the size of the jumps
jump_time_vect[0] = 0;
for (j = 1; j <= jump_number; j++)
  {
    jump_time_vect[j] = pnl_rand_uni_ab(0., T, generator);
    jump_size_vect[j] = delta * pnl_rand_normal(generator) + mean;
  }
//rearranging jump's times in ascending order
for (l = 0; l < jump_number; l++)
  {
    sup0 = jump_time_vect[0];
    for (j = 0; j < jump_number + 1 - l; j++)
      {
        if (jump_time_vect[j] > sup0)
          {
            sup0 = jump_time_vect[j];
            k = j;
          }
      }
    if (k != jump_number - l)
      {
        temp = jump_time_vect[jump_number - l];
        jump_time_vect[jump_number - l] = jump_time_vect[k];
        jump_time_vect[k] = temp;
      }
  }
// simulation of one Levy path
for (k = 1; k <= n_points; k++)
  {
    N[k] = 0;
    for (j = 1; j <= jump_number; j++)
      {
        if (jump_time_vect[j] <= t[k])
          N[k]++;
```

```
      }
    s0 = 0;
    for (j = N[k - 1] + 1; j <= N[k]; j++)
      s0 += jump_size_vect[j];
    X[k] = X[k - 1] + (W[k] - W[k - 1]) + s0;
  }
//computation of the supremum and the infimum of the Levy path and its
inf = X[0];
sup = X[0];
infw = W[0];
supw = W[0];
for (j = 1; j <= n_points; j++)
  {
    if (inf > X[j])
      inf = X[j];
    if (sup < X[j])
      sup = X[j];
    if (infw > W[j])
      infw = W[j];
    if (supw < W[j])
      supw = W[j];
  }
proba = 0;
exp_inf = S0 * exp(inf);
exp_infw = S0 * exp(infw);
if (exp_inf > s_shift)
  {
    exp_inf = s_shift;
    proba = 1.;
  }
if (exp_infw > s_shift)
  {
    exp_infw = s_shift;
  }
payoff = exp_inf;
control[0] = exp_infw;
exp_inf = S0 * exp(X[n_points] - sup); //antithetic variate associated
exp_infw = S0 * exp(W[n_points] - supw); //antithetic variate associat
if (exp_inf > s_shift)
  {
    exp_inf = s_shift;
```

```
            proba += 1.;
          }
        if (exp_infw > s_shift)
          {
            exp_infw = s_shift;
          }
        proba = proba / 2;
        payoff = discount * (payoff + exp_inf) / 2;
        control[0] = discount * ((control[0] + exp_infw) / 2);
        control[1] = discount * S0 * exp(X[n_points]);
        s1 += payoff;
        s2 += payoff * payoff;
        for (j = 0; j <= 1; j++)
          {
            s[j] += control[j];
            s3[j] += control[j] * payoff;
            for (k = 0; k <= 1; k++)
              s4[j][k] += control[j] * control[k];
          }
        s5 += proba;
        s6 += proba * proba;
      }
    for (j = 0; j <= 1; j++)
      {
        cov_payoff_control[j] = (s3[j] - s1 * s[j] / ((double)n_paths)) / (n_p
        for (k = 0; k <= 1; k++)
          {
            var_control[j][k] = (s4[j][k] - s[j] * s[k] / ((double)n_paths)) /
          }
      }
    determinant_control = var_control[0][0] * var_control[1][1] - var_control[
    coef_control[0] = (cov_payoff_control[0] * var_control[1][1] - cov_payoff_
    coef_control[1] = (cov_payoff_control[1] * var_control[0][0] - cov_payoff_
    var_payoff = (s2 - s1 * s1 / ((double)n_paths)) / (n_paths - 1);
    cor_payoff_control = sqrt((coef_control[0] * cov_payoff_control[0] + coef_
    var_proba = (s6 - s5 * s5 / ((double)n_paths)) / (n_paths - 1);
    *ptprice = -exp(-beta * sigma * sqrt(T / n_points)) * (s1 / n_paths - coef
    *priceerror = exp(-beta * sigma * sqrt(T / n_points)) * 1.96 * sqrt(var_pa
    *ptdelta = (*ptprice + discount * s_maxmin * s5 / n_paths) / S0;
    *deltaerror = (*priceerror + discount * s_maxmin * 1.96 * sqrt(var_proba)
  }
```

```
else//Put
  if ((P->Compute) == &Put_StrikeSpot2)
    {
      s_shift = exp(-beta * sigma * sqrt(T / n_points)) * s_maxmin;
      e1 = (log(S0 / s_maxmin) * (S0 < s_maxmin) + (nu + sigma * sigma) * T) /
      e2 = e1 - sigma * sqrt(T);
      e3 = (log(S0 / s_maxmin) * (S0 < s_maxmin) - nu * T) / (sigma * sqrt(T))
      //first control variable's expected value : it's the exponential of the
      if (S0 <= s_maxmin)
        expect_control_0 = exp(-beta * sigma * sqrt(T / n_points)) * (discount
      else
        expect_control_0 = exp(-beta * sigma * sqrt(T / n_points)) * (S0 * exp
      for (i = 0; i < n_paths; i++)
        {
          jump_number = pnl_rand_poisson(lambda * T, generator); // number of
          W[0] = 0;
          //simulation of the continuous part of the Levy : Brownian motion pa
          for (j = 1; j <= n_points; j++)
            {
              W[j] = sigma * pnl_rand_normal(generator) * sqrt(pas) + nu * pas
            }
          jump_time_vect[0] = 0;
          // simulation of the jump's times and the size of the jumps
          for (j = 1; j <= jump_number; j++)
            {
              jump_time_vect[j] = pnl_rand_uni_ab(0., T, generator);
              jump_size_vect[j] = delta * pnl_rand_normal(generator) + mean;
            }
          //rearranging jump's times in ascending order
          for (l = 0; l < jump_number; l++)
            {
              sup0 = jump_time_vect[0];
              for (j = 0; j < jump_number + 1 - l; j++)
                {
                  if (jump_time_vect[j] > sup0)
                    {
                      sup0 = jump_time_vect[j];
                      k = j;
                    }
                }
              if (k != jump_number - l)
```

```
      {
        temp = jump_time_vect[jump_number - 1];
        jump_time_vect[jump_number - 1] = jump_time_vect[k];
        jump_time_vect[k] = temp;
      }
  }
// simulation of one Levy path
for (k = 1; k <= n_points; k++)
  {
    N[k] = 0;
    for (j = 1; j <= jump_number; j++)
      {
        if (jump_time_vect[j] <= t[k])
          N[k]++;
      }
    s0 = 0;
    for (j = N[k - 1] + 1; j <= N[k]; j++)
      s0 += jump_size_vect[j];
    X[k] = X[k - 1] + (W[k] - W[k - 1]) + s0;
  }
//computation of the supremum and the infimum of the Levy path and i
sup = X[0];
inf = X[0];
infw = W[0];
supw = W[0];
for (j = 1; j <= n_points; j++)
  {
    if (inf > X[j])
      inf = X[j];
    if (sup < X[j])
      sup = X[j];
    if (infw > W[j])
      infw = W[j];
    if (supw < W[j])
      supw = W[j];
  }
proba = 0;
exp_sup = S0 * exp(sup);
exp_supw = S0 * exp(supw);
if (exp_sup < s_shift)
  {
```

```
            exp_sup = s_shift;
            proba = 1.;
          }
        if (exp_supw < s_shift)
          {
            exp_supw = s_shift;
          }
        payoff = exp_sup;
        control[0] = exp_supw;
        exp_sup = S0 * exp(X[n_points] - inf); //antithetic variat associate
        exp_supw = S0 * exp(W[n_points] - infw); //antithetic variate associ
        if (exp_sup < s_shift)
          {
            exp_sup = s_shift;
            proba += 1.;
          }
        if (exp_supw < s_shift)
          {
            exp_supw = s_shift;
          }
        proba = proba / 2.;
        payoff = discount * (payoff + exp_sup) / 2;
        control[0] = discount * (control[0] + exp_supw) / 2;
        control[1] = discount * S0 * exp(X[n_points]);
        s1 += payoff;
        s2 += payoff * payoff;
        for (j = 0; j <= 1; j++)
          {
            s[j] += control[j];
            s3[j] += control[j] * payoff;
            for (k = 0; k <= 1; k++)
              s4[j][k] += control[j] * control[k];
          }
        s5 += proba;
        s6 += proba * proba;
      }
    for (j = 0; j <= 1; j++)
      {
        cov_payoff_control[j] = (s3[j] - s1 * s[j] / ((double)n_paths)) / (n
        for (k = 0; k <= 1; k++)
          {
```

```
                var_control[j][k] = (s4[j][k] - s[j] * s[k] / ((double)n_paths))
              }
            }
          determinant_control = var_control[0][0] * var_control[1][1] - var_contro
          coef_control[0] = (cov_payoff_control[0] * var_control[1][1] - cov_payof
          coef_control[1] = (cov_payoff_control[1] * var_control[0][0] - cov_payof
          var_payoff = (s2 - s1 * s1 / ((double)n_paths)) / (n_paths - 1);
          cor_payoff_control = sqrt((coef_control[0] * cov_payoff_control[0] + coe
          var_proba = (s6 - s5 * s5 / ((double)n_paths)) / (n_paths - 1);
          *ptprice = exp(beta * sigma * sqrt(T / n_points)) * (s1 / n_paths - coef
          *priceerror = exp(beta * sigma * sqrt(T / n_points)) * 1.96 * sqrt(var_p
          *ptdelta = (*ptprice - discount * s_maxmin * s5 / n_paths) / S0;
          *deltaerror = (*priceerror + discount * s_maxmin * 1.96 * sqrt(var_proba
        }
    free(X);
    free(W);
    free(t);
    free(N);
    free(coef_control);
    free(control);
    free(s);
    free(s3);
    free(cov_payoff_control);
    for (i = 0; i <= 1; i++)
      {
        free(var_control[i]);
        free(s4[i]);
      }
    free(var_control);
    free(s4);
    free(jump_time_vect);
    free(jump_size_vect);
    return OK;
}

int CALC(MC_Merton_FloatingLookback)(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  Merton_Mc_FloatingLookback((ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[4].Va
                                     ptOpt->PayOff.Val.V_NUMFUNC_2,
                                     ptMod->S0.Val.V_PDOUBLE,
                                     ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V
                                     r, divid, ptMod->Sigma.Val.V_PDOUBLE,
                                     ptMod->Lambda.Val.V_PDOUBLE, ptMod->Mean.Va
                                     Met->Par[0].Val.V_ENUM.value, Met->Par[1].V
                                     &(Met->Res[0].Val.V_DOUBLE), &(Met->Res[1].
}

static int CHK_OPT(MC_Merton_FloatingLookback)(void *Opt, void *Mod)
{

  if ((strcmp(((Option *)Opt)->Name, "LookBackCallFloatingEuro") == 0) || (strcm
    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_PINT = 100;
      Met->Par[2].Val.V_LONG = 10000;
    }
  return OK;
}

PricingMethod MET(MC_Merton_FloatingLookback) =
{
  "MC_Merton_FloatingLookback",
  { {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Number of discretization steps", LONG, {100}, ALLOW}, {"N iterations", LON
  },
```

```
  CALC(MC_Merton_FloatingLookback),
  {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID}, {"Price E
  CHK_OPT(MC_Merton_FloatingLookback),
  CHK_ok,
  MET(Init)
} ;
```