Help

```
/* Glasserman-Heidelberger-Shahabuddin Algorithm
 Importance Sampling and Stratification Variance Reduction*/

#include <stdlib.h>
#include "
href../../../mod/bs1d/bs1d_pad/bs1d_pad_h_src.pdfbs1d_pad.h"
#include "
href../../../common/enums_h_src.pdfenums.h"

#define FACTOR  1.6
#define JMAX     40
#define NTRY     80



#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
static int CHK_OPT(MC_FixedAsian_StratificationAdaptive)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_FixedAsian_StratificationAdaptive)(void *Opt, void *Mod, PricingMeth
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

static double mu[50000];
static double t, sig, ri, dvd, S0, strike, step_nb;

/* Find the domain containg the zero of the function*/
static int zbrac(double(*func)(double), double *xmin, double *xmax)
{
  int j;
  double f1, f2;

  if (*xmin == *xmax)
    printf("mauvais depart dans la fonction zbrac()");

  f1 = (*func)(*xmin);
```

```
  f2 = (*func)(*xmax);

  for (j = 1; j <= NTRY; j++)
    {
      {
        if (f1 * f2 < 0.0)
          return 1;
      }

      if (fabs(f1) < fabs(f2))
        f1 = (*func)(*xmin += FACTOR * (*xmin - *xmax));
      else
        f2 = (*func)(*xmax += FACTOR * (*xmax - *xmin));
    }
  return 0; /*envoie 0 si [xmin,xmax] devient trop large*/
}

/*-----------------------------------------------------------------------*/
/* Methode de dichotomies permet de trouver un zero d'une fonction*/
/* sachant que ce zero se trouve entre x1 et x2. Precision = xacc*/
/*----------------------------------------------------------------------*/
static double rtbis(double (*func)(double), double x1, double x2, double xacc)
{
  int j;
  double dx, f, fmid, xmid, rtb;

  f = (*func)(x1);
  fmid = (*func)(x2);


  if (f * fmid >= 0.0)
    {
      printf("La racine ne se trouve pas dans [x1,x2]");
      exit(-1);
    }

  rtb = f < 0.0 ? (dx = x2 - x1, x1) : (dx = x1 - x2, x2); /* oriente la recherc

  for (j = 1; j <= JMAX; j++)
    {
      fmid = (*func)(xmid = rtb + (dx *= 0.5));
```

```
      if (fmid <= 0.0)rtb = xmid;
      if (fabs(dx) < xacc || fmid == 0.0)return rtb;
    }

  return 0.0;
}

/*--------------------------------------------------------------------------*/
/*Premiere partie : recherche du mu optimal*/
/*La fonction ci-dessous est celle qu'il faut appeller pour trouver le mu */
/*optimal. On cherche d'abord son unique racine qu'on reinjecte ensuite*/
/*dans les z[1..PAS] et s[1..PAS]; le dernier z[] est alors le mu optimal.*/
static double ghscall(double g)
{
  int i;
  double z = 0.0;
  double s;
  double dt, ans, s_dt, trend;

  s = S0;
  dt = t / step_nb;
  s_dt = sig * sqrt(dt);
  trend = (ri - dvd - 0.5 * sig * sig) * dt;

  if (g != 0)
    {
      ans = 0;
      z = s_dt * (g + strike) / g;
      for (i = 1; i < step_nb; i++)
        {
          s = s * exp(trend + s_dt * z);
          z = z - s_dt * s / (step_nb * g);
          ans += s;
        }

      ans /= step_nb;
      return (ans = (ans - strike - g));
    }
  return 0.0;
}
/*-----------------------------*/
```

```
static double ghsput(double g)
{
  int i;
  double z = 0.0;
  double s;
  double dt, ans, s_dt, trend;

  s = S0;
  dt = t / step_nb;
  s_dt = sig * sqrt(dt);
  trend = (ri - dvd - 0.5 * sig * sig) * dt;

  if (g != 0)
    {
      ans = s;
      z = s_dt * (g - strike) / g;
      for (i = 1; i < step_nb; i++)
        {
          s = s * exp(trend + s_dt * z);
          z = z + s_dt * s / (step_nb * g);
          ans += s;
        }

      ans /= step_nb;
      return (ans = (strike - ans - g));
    }
  else
    {
      printf("problem at line 138 of Pricin_util.h ...\ n");
      exit(-1);
    }
}


/* ------------------------------------------------------------------------ */
/* Computation of drift correction                                           */
/* ------------------------------------------------------------------------ */

static void Drift_Computation(int generator, int step_number, double T,  double
{
  double   S_t;
```

```
double h = T / step_number;
/* double sqrt_h = sqrt(h);*/
double trend = (r - divid) - 0.5 * SQR(sigma);
double ss_dt = sigma * sqrt(h);
double *xmin, *xmax, x_min, x_max, dot2;
int i;
double g;

t = T;
ri = r;
S0 = x;
strike = K;
sig = sigma;
dvd = divid;
step_nb = step_number;




for (i = 0; i < step_number; i++)
  mu[i] = 0.;

if ((p->Compute) == &Call_OverSpot2)
  {
    x_min = 2.5 * t;
    x_max = 5.0 * t;
    xmin = &x_min;
    xmax = &x_max;
    /*trouve le bon intervalle [xmin,xmax]*/
    zbrac(ghscall, xmin, xmax);
    /*resoud l equation ghs(x)=0*/
    g = rtbis(ghscall, (*xmin), (*xmax), 1e-8);
    mu[0] = ss_dt * (g + K) / g;
    dot2 = SQR(mu[0]);
    S_t = 1.0;
    for (i = 1; i < step_number; i++)
      {
        mu[i] = mu[i - 1] - ss_dt * S0 * S_t / (step_number * g);
        S_t = S_t * exp(trend * h + ss_dt * mu[i]);
        dot2 += SQR(mu[i]);
      }
```

```
    }
  else if ((p->Compute) == &Put_OverSpot2)
    {
      x_min = -5.0;
      x_max = -0.1;
      xmin = &x_min;
      xmax = &x_max;
      /*trouve le bon intervalle [xmin,xmax]*/
      zbrac(ghsput, xmin, xmax);
      /*resoud l equation ghs(x)=0*/
      g = rtbis(ghsput, (*xmin), (*xmax), 1e-8);
      mu[0] = ss_dt * (g - K) / g;
      dot2 = SQR(mu[0]);
      S_t = 1.0;
      for (i = 1; i < step_number; i++)
        {
          mu[i] = mu[i - 1] + ss_dt * S0 * S_t / (step_number * g);
          S_t = S_t * exp(trend * h + ss_dt * mu[i]);
          dot2 += SQR(mu[i]);
        }
    }
  return;

}

static int  FixedAsian_Stratification_adap(double s, double K, double time_spent
{
  long i, ipath, k;
  double price_sample, delta_sample, mean_price, mean_delta, var_delta;
  int init_mc;
  int simulation_dim;
  double alpha, z_alpha, dot1, dot2; /* inc=0.001;*/
  double integral, S_t, g1;
  double h = t / (double)M;
  double sqrt_h = sqrt(h);
  double trend = (r - divid) - 0.5 * SQR(sigma);
  int step_number = M;
  double norme_mu, uniform, Xi, dot3, val, temp;
  int i_strat;

  double *Y_t, *u_t, *gauss_vect, *mean_price_strata, *mean_sprice_strata;
```

```
double *sig_strata, *q, *mean_delta_strata, *var_delta_strata;
int *n, *m  ;
double Ss;/**/
int NB[3];   /*3 Steps*/


Y_t = malloc(M * sizeof(double));
u_t = malloc(M * sizeof(double));
gauss_vect = malloc(M * sizeof(double));

/*  Dynamic memory allocation */
mean_price_strata = malloc(nb_strat * sizeof(double));
mean_sprice_strata = malloc(nb_strat * sizeof(double));
sig_strata = malloc(nb_strat * sizeof(double));
q = malloc(nb_strat * sizeof(double));
n = malloc(nb_strat * sizeof(int));
m = malloc(nb_strat * sizeof(int));
mean_delta_strata = malloc(nb_strat * sizeof(double));
var_delta_strata = malloc(nb_strat * sizeof(double));


/* Computation of the number of drawings made at Step 0, 1 and 2    */
NB[0] = nb / 10;
NB[1] = nb / 2;
NB[2] = (int)(2 / 5) * nb;

/* Value to construct the confidence interval */
alpha = (1. - confidence) / 2.;
z_alpha = pnl_inv_cdfnor(1. - alpha);

/*Initialization*/
mean_price = 0.0;
mean_delta = 0.0;
var_delta = 0.0;

/* Size of the random vector we need in the simulation */
simulation_dim = M;


/* MC sampling */
init_mc = pnl_rand_init(generator, simulation_dim, 10000);
```

```
/* Test after initialization for the generator */
if (init_mc == OK)
  {


    /* Computation of the change of drift (importance sampling)  */
    (void)Drift_Computation(generator, M, t, s, r, divid, sigma, p, K);

    dot2 = 0;
    for (i = 0; i < step_number; i++)
      dot2 += mu[i] * mu[i];

    norme_mu = sqrt(dot2);

    for (i = 0; i < M; i++)
      u_t[i] = mu[i] / norme_mu; /*u_t of norm 1: projection direction for str

    /*Initialization   */

    for (i_strat = 0; i_strat < nb_strat; i_strat++)
      {
        mean_price_strata[i_strat] = 0.0;
        mean_sprice_strata[i_strat] = 0.0;
        mean_delta_strata[i_strat] = 0.0;
        var_delta_strata[i_strat] = 0.0;

      }


    /* Proportions at Step 0 and initialization of the N_i's */
    for (i_strat = 0; i_strat < nb_strat; i_strat++)
      {
        q[i_strat] = 1.0 / nb_strat;
        n[i_strat] = 0;

      }

    /*Nb: Step 0 is in the loop but proportions at Step 0 have been computed o
    for (k = 0; k < 3; k++)
      {
        /*Initialization of the sum of the sigma_i*/
```

```
Ss = 0.0;



for (i_strat = 0; i_strat < nb_strat; i_strat++)
  {

    /*Computation of the allocations at current step*/

    /**/
    m[i_strat] = NB[k] * q[i_strat] + 1; /*integer part plus one*/
    n[i_strat] += m[i_strat]; /*m[i] drawings in stratum i at current



    for (ipath = 1; ipath <= m[i_strat]; ipath++)
      {
        /* Begin of the m[i] iterations */
        g1 = pnl_rand_gauss(step_number, CREATE, 0, generator);
        uniform = pnl_rand_uni(generator);
        val = (i_strat + uniform) / nb_strat;
        Xi = pnl_inv_cdfnor(val); /* 1d-normal law conditioned to be i

        /*Simulation of Conditional Gaussian Law*/

        /*Computation  of u_t'g1 where g1 is a gaussian vector N(0,I)
        dot3 = 0.0;
        for (i = 0 ; i < step_number ; i++)
          {
            g1 = pnl_rand_gauss(step_number, RETRIEVE, i, generator);
            gauss_vect[i] = g1;
            dot3 += u_t[i] * g1;
          }


        /*Computation of mu'*Y_t where Y_t= gaussian vector/ the proje
        dot1 = 0.;
        for (i = 0 ; i < step_number ; i++)
          {
            temp = (Xi - dot3) * u_t[i] + gauss_vect[i];
            Y_t[i] = temp;
```

```
            dot1 += temp * mu[i];

          }

        /*Simulation of Stock and Average, ie: computation of step_num
        integral = 0.0;
        S_t = s;
        for (i = 0 ; i < step_number ; i++)
          {
            S_t *= exp(trend * h + sigma * sqrt_h * (Y_t[i] + mu[i]));
            integral += S_t;
          }

        /*value of: "S barre"(Y_t+mu)*exp(-mu'Y_t-0.5mu'mu)*/
        price_sample = (p->Compute)(p->Par, s, integral / (double)step

        /* Delta */
        if (price_sample > 0.0)
          delta_sample = (1 - time_spent) * (integral / (s * (double)s
        else  delta_sample = 0.;

        mean_price_strata[i_strat] += price_sample;
        mean_sprice_strata[i_strat] += SQR(price_sample);

        mean_delta_strata[i_strat] += delta_sample;
        var_delta_strata[i_strat] += SQR(delta_sample);

      }/*End of MonteCarlo, of the m[i] iterations*/



    /*Computation of empirical standard deviation in stratum i*/
    sig_strata[i_strat] = sqrt((mean_sprice_strata[i_strat] - SQR(mean

    Ss += sig_strata[i_strat];


  }
/* End of the nb_strat iterations*/
/*Computation of proportions at next step*/
```

```
    /*NB: The p_i disappear because strata have the same probability */

    /**/
    for (i_strat = 0; i_strat < nb_strat; i_strat++)
      {
        q[i_strat] = sig_strata[i_strat] / Ss;
      }


  }/*End of the 3 Steps*/

/*Computation of estimated value*/
for (i_strat = 0; i_strat < nb_strat; i_strat++)
  {
    mean_price += mean_price_strata[i_strat] / (double)n[i_strat];
  }


/* Delta*/

for (i_strat = 0; i_strat < nb_strat; i_strat++)
  {
    mean_delta += mean_delta_strata[i_strat] / (double)n[i_strat];
  }
/*Computation of empirical variances of Delta*/
for (i_strat = 0; i_strat < nb_strat; i_strat++)
  {
    var_delta_strata[i_strat] = (var_delta_strata[i_strat] - SQR(mean_delt
  }


/*For the computation of the Delta error */
for (i_strat = 0; i_strat < nb_strat; i_strat++)
  {
    /**/
    if (sig_strata[i_strat] > 0)
      var_delta += var_delta_strata[i_strat] / (double)nb_strat / sig_stra
  }


/**/
```

```
      *ptprice = (mean_price / (double)nb_strat);
      *ptprice = exp(-r * t) * (*ptprice);
      /* Price Confidence Interval */
      /* Variance is computed as (\ sum p_i*sig_i)^2/nb with sig_i the estimated
      *pterror_price = exp(-r * t) * Ss / (double)nb_strat / sqrt(nb);

      *inf_price = *ptprice - z_alpha * (*pterror_price);
      *sup_price = *ptprice + z_alpha * (*pterror_price);

      /* Price estimator */



      /* Delta estimator */
      *ptdelta = exp(-r * t) * (mean_delta / (double)nb_strat);
      if ((p->Compute) == &Put_OverSpot2)
        *ptdelta *= (-1);
      *pterror_delta = exp(-r * t) * sqrt(Ss * var_delta / (double)nb_strat / nb

      /* Delta Confidence Interval */
      *inf_delta = *ptdelta - z_alpha * (*pterror_delta);
      *sup_delta = *ptdelta + z_alpha * (*pterror_delta);


    }

  free(Y_t);
  free(u_t);
  free(gauss_vect);
  free(mean_price_strata);
  free(mean_sprice_strata);
  free(sig_strata);
  free(q);
  free(n);
  free(m);
  free(mean_delta_strata);
  free(var_delta_strata);
  return init_mc;
}

/*****************************************************************************
```

```
int CALC(MC_FixedAsian_StratificationAdaptive)(void *Opt, void *Mod, PricingMeth
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;

  double T, t_0, T_0;
  double r, divid, time_spent, pseudo_strike, true_strike, pseudo_spot;
  int return_value;

  r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
  divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

  T = ptOpt->Maturity.Val.V_DATE;
  T_0 = ptMod->T.Val.V_DATE;
  t_0 = (ptOpt->PathDep.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE;
  time_spent = (T_0 - t_0) / (T - t_0);

  if (T_0 < t_0)
    {
      Fprintf(TOSCREEN, "T_0 < t_0, untreated case\ n\ n\ n");
      return_value = WRONG;
    }

  /* Case t_0 <= T_0 */
  else
    {
      pseudo_spot = (1. - time_spent) * ptMod->S0.Val.V_PDOUBLE;
      pseudo_strike = (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE - ti


      true_strike = (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE;

      (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE = pseudo_strike;

      if (pseudo_strike <= 0.)
        {
          Fprintf(TOSCREEN, "FORMULE ANALYTIQUE\ n\ n\ n");
          return_value = Analytic_KemnaVorst(pseudo_spot,
                                             pseudo_strike,
                                             time_spent,
                                             ptOpt->PayOff.Val.V_NUMFUNC_2,
```

```
                                                        T - T_0,
                                                        r,
                                                        divid,
                                                        &(Met->Res[0].Val.V_DOUBLE),
                                                        &(Met->Res[1].Val.V_DOUBLE));

        }
      else
        return_value = FixedAsian_Stratification_adap(pseudo_spot,
                       pseudo_strike,
                       time_spent,
                       ptOpt->PayOff.Val.V_NUMFUNC_2,
                       T - T_0,
                       r,
                       divid,
                       ptMod->Sigma.Val.V_PDOUBLE,
                       Met->Par[3].Val.V_LONG,
                       Met->Par[1].Val.V_INT2,
                       Met->Par[0].Val.V_INT2,
                       Met->Par[2].Val.V_ENUM.value,
                       Met->Par[4].Val.V_DOUBLE,
                       &(Met->Res[0].Val.V_DOUBLE),
                       &(Met->Res[1].Val.V_DOUBLE),
                       &(Met->Res[2].Val.V_DOUBLE),
                       &(Met->Res[3].Val.V_DOUBLE),
                       &(Met->Res[4].Val.V_DOUBLE),
                       &(Met->Res[5].Val.V_DOUBLE),
                       &(Met->Res[6].Val.V_DOUBLE),
                       &(Met->Res[7].Val.V_DOUBLE));

      (ptOpt->PayOff.Val.V_NUMFUNC_2)->Par[0].Val.V_PDOUBLE = true_strike;
    }
  return return_value;
}



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

```
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  int type_generator;
  if (Met->init == 0)
    {
      Met->init = 1;

      Met->Par[0].Val.V_INT2 = 360;
      Met->Par[1].Val.V_INT2 = 100;
      Met->Par[2].Val.V_ENUM.value = 0;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumRNGs;
      Met->Par[3].Val.V_LONG = 20000;
      Met->Par[4].Val.V_DOUBLE = 0.95;


    }

  type_generator = Met->Par[2].Val.V_ENUM.value;

  if (pnl_rand_or_quasi(type_generator) == PNL_QMC)
    {
      Met->Res[2].Viter = IRRELEVANT;
      Met->Res[3].Viter = IRRELEVANT;
      Met->Res[4].Viter = IRRELEVANT;
      Met->Res[5].Viter = IRRELEVANT;
      Met->Res[6].Viter = IRRELEVANT;
      Met->Res[7].Viter = IRRELEVANT;

    }
  else
    {
      Met->Res[2].Viter = ALLOW;
      Met->Res[3].Viter = ALLOW;
      Met->Res[4].Viter = ALLOW;
      Met->Res[5].Viter = ALLOW;
      Met->Res[6].Viter = ALLOW;
      Met->Res[7].Viter = ALLOW;
    }
```

```
  return OK;
}


PricingMethod MET(MC_FixedAsian_StratificationAdaptive) =
{
  "MC_FixedAsian_Stratification_Adaptive",
  { {"TimeStepNumber", INT2, {100}, ALLOW},
    {"Number of Strata", INT2, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Total Number of iterations", LONG, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_FixedAsian_StratificationAdaptive),
  { {"Price", DOUBLE, {100}, FORBID},
    {"Delta", DOUBLE, {100}, FORBID} ,
    {"Error Price", DOUBLE, {100}, FORBID},
    {"Error Delta", DOUBLE, {100}, FORBID} ,
    {"Inf Price", DOUBLE, {100}, FORBID},
    {"Sup Price", DOUBLE, {100}, FORBID} ,
    {"Inf Delta", DOUBLE, {100}, FORBID},
    {"Sup Delta", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_FixedAsian_StratificationAdaptive),
  CHK_ok,
  MET(Init)
};
```