Help

```
/* Céline LABART (CMAP) and Jérôme LELONG (CERMICS), September 2006
 * Pricing of double barrier parisian options using Laplace
 * Transforms
 * Options considered: Knock in or out Calls or Puts
 */

extern "C" {
#include "bs1d_doublim.h"
}
#include <complex>
#include <cmath>

using namespace std;

typedef complex<double>          complex_double;
#define LOWER 0
#define UPPER 1

/** \ defgroup Double Double barrier
 * \ struct parisian_double_t
 * Structure used to describe a Double barrier Parisian
 * option */
typedef struct
{
  double K; /*!< Strike  */
  double T; /*!< Maturity  */
  double t; /*!< Pricing time: not implmented for double
              barrier */
  double D; /*!< Delay */
  double d; /*!< time already spent in the excursion when
             * pricing at time t>0 : not implemented yet */
  double L; /*!< lower barrier */
  double U; /*!< upper barrier */
  double sigma; /*!< volatility */
  double r; /*!< Instantaneous Interest Rate*/
  double delta; /*!< Instantaneous Dividend Rate*/
  double So; /*!< Spot */
} parisian_double_t;
```

```
/** \ addtogroup Double
 * @{
 * Constructor for parisian_double_t
 * Allocates a new instance.
 * \ param orig is used to initialise the new structure
 * \ return a pointer on this instance */
static parisian_double_t *NewParisian_Double_t(const parisian_double_t *orig)
{
  parisian_double_t *opt = new parisian_double_t;
  opt->K = orig->K;
  opt->T = orig->T;
  opt->t = orig->t;
  opt->D = orig->D;
  opt->d = orig->d;
  opt->L = orig->L;
  opt->U = orig->U;
  opt->sigma = orig->sigma;
  opt->r = orig->r;
  opt->So = orig->So;
  opt->delta = orig->delta;
  return opt;
}

/* @} */

/** \ defgroup Single Single barrier
 * \ struct parisian_t
 * Structure used to describe a Singlebarrier Parisian
 * option */
typedef struct
{
  double K; /*!< Strike  */
  double T; /*!< Maturity  */
  double t; /*!< Pricing time: not implmented for double
              barrier */
  double D; /*!< Delay */
  double d; /*!< time already spent in the excursion when
             * pricing at time t>0 */
  double L; /*!< barrier */
  double sigma; /*!< volatility */
  double r; /*!< Instantaneous Interest Rate*/
```

```
  double delta; /*!< Instantaneous Dividend Rate*/
  double So; /*!< Spot */
} parisian_t;

/** \ addtogroup Single
 * @{
 * Constructor for parisian_t
 * Allocates a new instance.
 * \ param orig is used to initialise the new structure
 * \ return a pointer on this instance */
static parisian_t *NewParisian_t(const parisian_t *orig)
{
  parisian_t *opt = new parisian_t;
  opt->K = orig->K;
  opt->T = orig->T;
  opt->t = orig->t;
  opt->D = orig->D;
  opt->d = orig->d;
  opt->L = orig->L;
  opt->sigma = orig->sigma;
  opt->r = orig->r;
  opt->So = orig->So;
  opt->delta = orig->delta;
  return opt;
}

/** Creates an instance of parisian_t from parisian_double_t
 * Allocates a new instance, initialises it with orig and
 * \ return a pointer on this instance.
 * \ param upperOrLower tells if the barrier of the single option is
 the lower (0) or the upper (1) barrier of the double option*/
static parisian_t *NewParisian_FromDouble(const parisian_double_t *orig,
    int upperOrLower)
{
  parisian_t *opt = new parisian_t;
  opt->K = orig->K;
  opt->T = orig->T;
  opt->t = orig->t;
  opt->D = orig->D;
  opt->d = orig->d;
```

```
  if (upperOrLower == UPPER)
    opt->L = orig->U;
  else
    opt->L = orig->L;

  opt->sigma = orig->sigma;
  opt->r = orig->r;
  opt->So = orig->So;
  opt->delta = orig->delta;
  return opt;
}

/** @} */

/* defined in Src/common/complex_erf.c */
extern complex_double normal_cerf(const complex_double  z);


static complex_double psi(complex_double z)
{
  complex_double res;
  double racine = sqrt(2.0 * M_PI);
  res = 1.0 + z * racine * exp(z * z / 2.0) * normal_cerf(z);
  return (res);
}

/** Laplace transform of Z_{T_b^-}
 * @param l : laplace parameter
 * @param b : barrier
 * @param D : length of the excursion
 *
 * @returns Laplace transform of Z_{T_b^-}
 */
static complex_double Laplace_Z_T_b_minus(complex_double l, double b, double D)
{
  double d = sqrt(D);
  if (b < 0)
    return exp(-l * b) * psi(l * d);

  return 2.0 * normal_cerf(-b / d) * exp(-l * b) * psi(l * d) +
         exp(l * l / 2.0 * D) * (normal_cerf(b / d + l * d) - exp(-2.0 * l * b)
```

```
}

/** Laplace transform of T_b^-
 * @param l : laplace parameter
 * @param b : barrier
 * @param D : length of the excursion
 *
 * @returns Laplace transform of T_b^-
 */
static complex_double Laplace_T_b_minus(complex_double l, double b, double D)
{
  complex_double theta = sqrt(2.0 * l);
  if (b < 0)
    return exp(theta * b) / psi(theta * sqrt(D));

  return exp(-l * D) * (1.0 - 2.0 * normal_cerf(-b / sqrt(D))) +
         (exp(-theta * b) * normal_cerf(theta * sqrt(D) - b / sqrt(D)) +
          exp(theta * b) * normal_cerf(-theta * sqrt(D) - b / sqrt(D))) / psi(th
}


/** Laplace transform of Z_{T_b^+}
 * @param l : laplace parameter
 * @param b : barrier
 * @param D : length of the excursion
 *
 * @returns Laplace transform of Z_{T_b^+}
 */
static complex_double Laplace_Z_T_b_plus(complex_double l, double b, double D)
{
  double d = sqrt(D);
  if (b > 0)
    return exp(-l * b) * psi(-l * d);

  return 2.0 * normal_cerf(b / d) * exp(-l * b) * psi(-l * d) +
         exp(l * l / 2.0 * D) * (normal_cerf(-b / d - l * d) - exp(-2.0 * l * b)
}


/** Laplace transform of T_b^+
 * @param l : laplace parameter
```

```
 * @param b : barrier
 * @param D : length of the excursion
 *
 * @returns Laplace transform of T_b^+
 */
static complex_double Laplace_T_b_plus(complex_double l, double b, double D)
{
  complex_double theta = sqrt(2.0 * l);
  if (b > 0)
    return exp(-theta * b) / psi(theta * sqrt(D));

  return exp(-l * D) * (1.0 - 2.0 * normal_cerf(b / sqrt(D))) +
         (exp(theta * b) * normal_cerf(theta * sqrt(D) + b / sqrt(D)) +
          exp(-theta * b) * normal_cerf(-theta * sqrt(D) + b / sqrt(D))) / psi(t
}

/**
 * @param l : laplace parameter
 * @param b1 : lower barrier
 * @param b2 : upper barrier
 * @param D : length of the excursion
   \ f[
   E (\ exp(- \ |ambda T_{b_1}^-) {\ bf 1}_{\ {T_{b_1}^- <
   T_{b_2}^+\ }})
   \ f]
*/
static complex_double Laplace_T_b_minus_ind(complex_double l, double b1, double
{
  complex_double theta = sqrt(2.0 * l);
  complex_double a1 = exp(theta * b1) / psi(theta * sqrt(D)) * Laplace_Z_T_b_plu
  complex_double a2 = exp(-theta * b2) / psi(theta * sqrt(D)) * Laplace_Z_T_b_mi
  return (Laplace_T_b_minus(l, b1, D) - a1 * Laplace_T_b_plus(l, b2, D)) / (1.0
}


/**
   \ f[
   E (\ exp(- \ |ambda T_{b_2}^+) {\ bf 1}_{\ {T_{b_2}^+ <
   T_{b_1}^-\ }})
   \ f]
*/
```

```
static complex_double Laplace_T_b_plus_ind(complex_double l, double b1, double b
{
  complex_double theta = sqrt(2.0 * l);
  complex_double a1 = exp(theta * b1) / psi(theta * sqrt(D)) * Laplace_Z_T_b_plu
  complex_double a2 = exp(-theta * b2) / psi(theta * sqrt(D)) * Laplace_Z_T_b_mi
  return (Laplace_T_b_plus(l, b2, D) - a2 * Laplace_T_b_minus(l, b1, D)) / (1.0
}
```

```
static complex_double pdic(complex_double l, const parisian_t *opt);
static complex_double puic(complex_double l, const parisian_t *opt);
static complex_double pdic_L_x(complex_double l, const parisian_t *opt);
static complex_double puic_x_L(complex_double l, const parisian_t *opt);
static complex_double puoc(complex_double l, const parisian_t *opt);
static complex_double bs(complex_double l, const parisian_t *opt);
```

```
/** \ addtogroup Single
 * @{
 * Laplace transform of the price of the bs call with respect
 * to maturity time */
static complex_double bs(complex_double l, const parisian_t *opt)
{
  complex_double theta;
  double  m, k;
  m = (opt->r - opt->delta - pow(opt->sigma, 2.0) / 2.0) / opt->sigma;
  k = log(opt->K / opt->So) / opt->sigma;
  theta = sqrt(2.0 * l);

  /* K < X */
  if (opt->K <= opt->So)
    return (2.0 * opt->K / (m * m - 2.0 * l) - 2.0 * opt->So / (pow(m + opt->sig
            opt->K * exp((m + theta) * k) / theta * (1.0 / (m + theta) - 1.0 / (
  /*K > x*/
  else
    return (opt->K * exp((m - theta) * k) / theta * (1.0 / (m - theta) - 1.0 / (

  return WRONG;
}
```

```
/** Laplace transform of a Down In Call  for L<x*/
static complex_double pdic_L_x(complex_double l, const parisian_t *opt)
{
  complex_double theta;
  double m;
  double b;
  double k;
  double d;
  double d3;
  double racine = sqrt(2.0 * M_PI);
  m = 1.0 / opt->sigma * (opt->r - opt->delta - pow(opt->sigma, 2.0) / 2.0);
  b = 1.0 / opt->sigma * log(opt->L / opt->So);
  k = 1.0 / opt->sigma * log(opt->K / opt->So);
  theta = sqrt(2.0 * l);
  d = sqrt(opt->D);
  d3 = (b - k) / d;
  if (opt->K <= opt->L)
    {
      return (exp((m + theta) * b) / psi(theta * d) * (2.0 * opt->K / (m * m - 2

  /*L<K*/
  if (opt->L <= opt->K)
    {
      return (psi(-theta * d) / psi(theta * d) * opt->K / theta * exp(2.0 * b *
    }
  return WRONG;
}



/** Laplace transform of a single Parisian Up Out Call */
static complex_double puoc(complex_double l, const parisian_t *opt)
{
  complex_double theta;
  parisian_t *opt_0 = NewParisian_t(opt);
  double  b, m, d;
  complex_double res = 0;
  m = (opt->r - opt->delta - pow(opt->sigma, 2.0) / 2.0) / opt->sigma;
  b = log(opt->L / opt->So) / opt->sigma;
```

```
  theta = sqrt(2.0 * l);
  d = sqrt(opt->D);
  if (opt->D > opt->T)
    res = bs(l, opt);
  else
    {
      if (opt->L >= opt->So)
        res = bs(l, opt) - puic(l, opt);
      else
        {
          opt_0->K = opt->K / opt->L;
          opt_0->So = 1.0;
          opt_0->T = opt->T;
          opt_0->L = 1.0;
          opt_0->D = opt->D;
          opt_0->sigma = opt->sigma;
          opt_0->r = opt->r;
          opt_0->delta = opt->delta;

          res = opt->L * (exp((m + theta) * b) * normal_cerf(theta * d + b / d)
                * puoc(l, opt_0);
        }
    }
  delete opt_0;
  return res;
}


/* Parisian Up In call formula for x<L */
static complex_double puic_x_L(complex_double l, const parisian_t *opt)
{
  complex_double theta;
  double  b, m, k, d, d3;
  double racine = sqrt(2.0 * M_PI);
  m = (opt->r - opt->delta - pow(opt->sigma, 2.0) / 2.0) / opt->sigma;
  b = log(opt->L / opt->So) / opt->sigma;
  k = log(opt->K / opt->So) / opt->sigma;
  d = sqrt(opt->D);
  theta = sqrt(2.0 * l);
  d3 = (b - k) / d;
  if (opt->D > opt->T) return 0.0;
```

```
  /*  X < L < K*/
  if (opt->L < opt->K)
    return (exp((m - theta) * b) * racine * d / psi(theta * d) * (2.0 * opt->K /

  /* K<L and x<L*/
  if (opt->K <= opt->L)
    return (exp((m - theta) * b) / psi(theta * d) * (2.0 * opt->K / (m * m - 2.0

  return WRONG;
}

/** Laplace transform of a Up In Call */
static complex_double puic(complex_double l, const parisian_t *opt)
{
  complex_double theta;
  if (opt->D > opt->T) return 0.0;
  /*  x < L */
  if (opt->So <= opt->L)
    return puic_x_L(l, opt);

  /* x>L  */
  if (opt->So > opt->L)
    return (bs(l, opt) - puoc(l, opt));

  return WRONG;
}

/*************************************************************/

static complex_double pdic(complex_double l, const parisian_t *opt)
{
  complex_double theta;
  double m;
  double b;
  double k;
  double d;
  double racine = sqrt(2.0 * M_PI);
  m = 1.0 / opt->sigma * (opt->r - opt->delta - pow(opt->sigma, 2) / 2.0);
  b = 1.0 / opt->sigma * log(opt->L / opt->So);
  k = 1.0 / opt->sigma * log(opt->K / opt->So);
  theta = sqrt(2.0 * l);
```

```
  d = sqrt(opt->D);
  if (opt->D > opt->T) return 0.0;
  /*L<x*/
  if (opt->L <= opt->So)
    return pdic_L_x(l, opt);
  /*x<L<K*/
  if (opt->So <= opt->L)
    {
      return (opt->K / theta * exp((m - theta) * k) * (1.0 / (m - theta) - 1.0 /
    }

  /*x<K<L*/
  if (opt->So <= opt->K && opt->K <= opt->L)
    {
      return (opt->K / theta * exp((m - theta) * k) * (1.0 / (m - theta) - 1.0 /
    }
  /*K<x<L*/
  if (opt->K <= opt->So && opt->So <= opt->L)
    {
      return (2.0 * opt->K / (m * m - 2.0 * l) - 2.0 * opt->So / (pow(m + opt->s
    }

  return -1;
}


static complex_double pdoc(complex_double l, const parisian_t *opt)
{
  complex_double theta;
  double  b, m, k, d;
  double racine = sqrt(2.0 * M_PI);

  m = (opt->r - opt->delta - pow(opt->sigma, 2) / 2.0) / opt->sigma;
  b = log(opt->L / opt->So) / opt->sigma;
  k = log(opt->K / opt->So) / opt->sigma;
  d = sqrt(opt->D);
  theta = sqrt(2.0 * l);


  /* L < K  < X */
  if (opt->D > opt->T) return (bs(l, opt));
```

```
  if (opt->L <= opt->K && opt->K <= opt->So)
    return (2.0 * opt->K / (m * m - 2.0 * l) - 2.0 * opt->So / (pow(m + opt->sig

  /* L  < x < K */
  if (opt->L <= opt->So && opt->So <= opt->K)
    return ((1.0 - exp(2.0 * b * theta) + (theta * exp(2.0 * b * theta) * racine

  /* x < L< K */
  if (opt->So <= opt->L && opt->L <= opt->K)
    return (opt->L * (exp((m - theta) * b) * normal_cerf(theta * d - b / d) + ex

  /* K < L < x */
  if (opt->K <= opt->L && opt->L <= opt->So)

    return (2.0 * opt->K / (m * m - 2.0 * l) * (1.0 - exp((m + theta) * b) / psi


  /* K<L and x<L */
  if (opt->K <= opt->L && opt->So <= opt->L)
    return (opt->L * (exp((m - theta) * b) * normal_cerf(theta * d - b / d) + ex

  return -1;
}



/*****************************************************************/


/** @} */


/** \ addtogroup Double
 * @{
 Computes
 \ f[
 A_1.0= \ mathbf{E}\ left[1.0_{T_{b_1.0}^- < T} \ mathbf{E}\ left[ 1.0_{T_{b_1.0
 e^{m Z_T} (xe^{\ sigma Z_T} - K)_+|\ mathcal{F}_{T_{b_1.0}^-}\ right] \ right],
 \ f]
 * b1 brownian barrier corresponding to L
 * b2 brownian barrier corresponding to U */
```

```
static complex_double factor_A1(complex_double l,  const parisian_double_t *opt)
{
  complex_double A1;
  complex_double theta;
  complex_double E1, laplace_Z;
  double b1, b2;
  parisian_t *single;

  single = NewParisian_FromDouble(opt, UPPER);
  A1 = puic_x_L(l, single);
  delete single;


  theta = sqrt(2.0 * l);
  b1 = log(opt->L / opt->So) / opt->sigma;
  b2 = log(opt->U / opt->So) / opt->sigma;
  E1 = Laplace_T_b_minus_ind(l, b1, b2, opt->D);
  laplace_Z = Laplace_Z_T_b_minus(-theta, b1, opt->D);


  A1 *= E1 * laplace_Z;
  return A1;
}


/**  Computes
     \ f[
     A_2 =\ mathbf{E}\ left[1.0_{T_{-b_1.0}^+(\ tilde{Z}) < T}1.0_{T_{-b_2.0^-}(
     <T}e^{-(m+\ sigma)\ tilde{Z}_T}(x-Ke^{\ sigma\ tilde{Z}_T})_+\ right]
     \ f]
     * \ param b1 brownian barrier corresponding to L
     * \ param b2 brownian barrier corresponding to U */
static complex_double factor_A2(complex_double l,  const parisian_double_t *opt)
{
  complex_double A2, E2, laplace_Z;
  complex_double theta;
  double b1, b2;
  parisian_t *single;

  single = NewParisian_FromDouble(opt, LOWER);
  A2 = pdic_L_x(l, single);
```

```
  delete single;


  theta = sqrt(2.0 * l);
  b1 = log(opt->L / opt->So) / opt->sigma;
  b2 = log(opt->U / opt->So) / opt->sigma;
  E2 = Laplace_T_b_plus_ind(l, b1, b2, opt->D);
  laplace_Z = Laplace_Z_T_b_plus(theta, b2, opt->D);


  A2 *= E2 * laplace_Z;
  return A2;
}


/** Laplace Transform of the price of a double barrier
 * parisian knock in option */
static complex_double LKnockIn(complex_double l,  const parisian_double_t *opt)
{
  complex_double A1, A2;
  parisian_t *single;
  complex_double down_in, up_in;

  single = NewParisian_FromDouble(opt, LOWER);
  down_in = pdic(l, single);
  delete single;
  single = NewParisian_FromDouble(opt, UPPER);
  up_in = puic(l, single);
  delete single;

  A1 = factor_A1(l, opt);
  A2 = factor_A2(l, opt);

  return (down_in + up_in - A1 - A2);
}




/** Laplace Transform of the price of a double barrier
 * parisian knock out option */
```

```
static complex_double LKnockOut(complex_double l,  const parisian_double_t *opt)
{
  complex_double A1, A2;
  parisian_t *single;
  complex_double call_bs, down_in, up_in, down_out, up_out;

  single = NewParisian_FromDouble(opt, LOWER);
  call_bs = bs(l, single);
  down_in = pdic(l, single); /* PDIC(L_1) */
  down_out = pdoc(l, single); /* PDoC(L_1) */
  delete single;
  single = NewParisian_FromDouble(opt, UPPER);
  up_in = puic(l, single); /* PUIC(L_2) */
  up_out = puoc(l, single); /* PUOC(L_2) */
  delete single;

  A1 = factor_A1(l, opt);
  A2 = factor_A2(l, opt);

  /*  return (call_bs - down_in - up_in + A1+A2);*/
  return (up_out + down_out - call_bs  + A1 + A2);
}

/** @} */

/** Computes the numerical inversion of the Laplace transform
 * given as the first argument using Euler summation.
 *
 * The integral is approximated using a trapezoidal rule
 * with step <tt>h</tt> and the  non finite series is
 * computed using Euler acceleration. The optimal value is
 * \ f$h=\ pi/T\ f$. This choice enables some
 * simplification. The function below must NOT be used with
 * an other value for the step.
 *
 * \ param *f is the Laplace
 * transform to invert \ param opt describes the option
 * \ return the prices of the option <tt>opt</tt> at time 0
 *
 * Numerical implementation:
 * \ f[
```

```
 * E(m,n,T)= \ sum_{k=0}^{m}C_m^k 2.0^{-m}s_{n+k}(T)
 * \ f]
 * and
 * \ f[
 * s_n(t)= \ frac{e^{\ alpha t}}{2.0t} \ widehat{f}(\ alpha) +
 * \ frac{e^{\ alpha t}}{t} \ sum_{k=1.0}^{ n} (-1.0)^k \  \ mathop
 * {\ mathcal{R}\ mathrm{e}}\ left( \ widehat{f}\ left(\ alpha + i\ frac{\ pi k}
 * \ f]
 */
static double
euler(complex_double(*f)(complex_double l, const parisian_double_t *opt),
      const parisian_double_t *opt, int N, int M)
{
  int n, k, Cnp;
  double  sum, alpha, h, run_sum, m;
  /* int N=35 ;
   * int M=15; */
  double A;
  complex_double I = complex_double(0.0, 1.0);

  m = (opt->r - opt->delta - opt->sigma * opt->sigma / 2.0) / opt->sigma;

  A = 14; /*MAX(1.03, pow(m + opt->sigma,2.0)*opt->T+1.0);*/
  alpha = A / (2.0 * opt->T);
  h = M_PI / opt->T;
  run_sum = 0.5 * ((*f)(alpha, opt)).real();

  for (n = 1; n <= N; n++)
    run_sum = run_sum + PNL_ALTERNATE(n) * ((*f)(alpha + h * n * I, opt)).real()

  sum = run_sum; /*partial exponential sum */
  Cnp = 1; /* binomial coefficients */

  for (k = 1; k <= M; k++)
    {
      Cnp = (Cnp * (M - k + 1)) / k;
      run_sum = run_sum + PNL_ALTERNATE(N + k) * ((*f)(alpha + h * (N + k) * I,
      sum = sum + run_sum * (double) Cnp ;
    }
  return (exp(-(opt->r + m * m / 2.0) * opt->T) * exp(alpha * opt->T) * sum / po
}
```

```
/** \ addtogroup Double
 * 1.0 -> Call In
 * 2.0 -> Call Out
 * 3 -> Put In
 * 4 -> Put Out
 * Computes the price of the corresponding single barrier
 * Parisian option using Laplace inversion*/
static double
DoubleParisian(int choice, const parisian_double_t *opt, int N, int M)
{
  double res = 0.0;
  parisian_double_t *new_opt;
  switch (choice)
    {
    case 1:
      res = euler(LKnockIn, opt, N, M);
      break;
    case 2:
      res = euler(LKnockOut, opt, N, M);
      break;
    case 3:
    {
      new_opt = NewParisian_Double_t(opt);
      new_opt->So = 1.0 / opt->So;
      new_opt->U = 1.0 / opt->L;
      new_opt->L = 1.0 / opt->U;
      new_opt->K = 1.0 / opt->K;
      new_opt->r = opt->delta;
      new_opt->delta = opt->r;
      res = opt->K * opt->So * DoubleParisian(1, new_opt, N, M);
      delete new_opt;
    }
    break;
    case 4:
    {
      new_opt = NewParisian_Double_t(opt);
      new_opt->So = 1.0 / opt->So;
      new_opt->U = 1.0 / opt->L;
```

```
      new_opt->L = 1.0 / opt->U;
      new_opt->K = 1.0 / opt->K;
      new_opt->r = opt->delta;
      new_opt->delta = opt->r;
      res = opt->K * opt->So * DoubleParisian(2, new_opt, N, M);
      delete new_opt;
    }
    break;
    default:
    {
      printf("wrong choice in DoubleParisian\ n");
    }
    }
  return res;
}


static int
LaplaceDoubleParisian(int outorin, int callorput, double K, double s,
                      double t, double L, double U, double delay,
                      double r, double divid, double sigma,
                      double inc, int N, int M,
                      double *ptprice, double *ptdelta)
{

  parisian_double_t *opt = new parisian_double_t;
  int choice;

  opt->T = t;
  opt->t = 0.0;
  opt->D = delay;
  opt->r = r;
  opt->sigma = sigma;
  opt->delta = divid;
  opt->So = s;
  opt->L = L;
  opt->U = U;
  opt->d = 0.0;
  opt->K = K;
```

```
  if (!callorput)
    {
      /* out put */
      if (outorin)
        choice = 4;
      else
        /* in put */
        choice = 3;
    }
  else
    {
      /* out call */
      if (outorin)
        choice = 2;
      else
        /* in call */
        choice = 1;
    }

  /*Price*/
  *ptprice = DoubleParisian(choice, opt, N, M);

  /*Delta*/
  opt->So = opt->So * (1.0 + inc);
  *ptdelta = (DoubleParisian(choice, opt, N, M) - *ptprice) / (s * inc);

  delete opt;

  return OK;
}



extern "C"   {

  int CALC(AP_LaplaceDoubleParisian)(void *Opt, void *Mod, PricingMethod *Met)
  {
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r, divid;
    int outorin, callorput;
```

```
  r = log(1.0 + ptMod->R.Val.V_DOUBLE / 100.);
  divid = log(1.0 + ptMod->Divid.Val.V_DOUBLE / 100.);

  if ((ptOpt->PayOff.Val.V_NUMFUNC_1->Compute) == &Put)
    callorput = 0; /* Put */
  else
    callorput = 1;
  if ((ptOpt->OutOrIn).Val.V_BOOL == OUT)
    outorin = 1; /* out */
  else outorin = 0;



  return LaplaceDoubleParisian(outorin, callorput,
                               ptOpt->PayOff.Val.V_NUMFUNC_1->Par[0].Val.V_DOU
                               ptMod->S0.Val.V_PDOUBLE,
                               ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DAT
                               (ptOpt->LowerLimit.Val.V_NUMFUNC_1)->Par[0].Val
                               (ptOpt->UpperLimit.Val.V_NUMFUNC_1)->Par[0].Val
                               (ptOpt->LowerLimit.Val.V_NUMFUNC_1)->Par[1].Val
                               r,
                               divid,
                               ptMod->Sigma.Val.V_PDOUBLE,
                               Met->Par[0].Val.V_PDOUBLE,
                               Met->Par[1].Val.V_PINT,
                               Met->Par[2].Val.V_PINT,
                               &(Met->Res[0].Val.V_DOUBLE),
                               &(Met->Res[1].Val.V_DOUBLE));
}



static int CHK_OPT(AP_LaplaceDoubleParisian)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;
  TYPEOPT *opt = (TYPEOPT *)(ptOpt->TypeOpt);

  if ((opt->TwoDoubleStep).Val.V_BOOL == FALSE)
    if ((opt->RebOrNo).Val.V_BOOL == NOREBATE)
      if ((opt->EuOrAm).Val.V_BOOL == EURO &&
```

```
          (opt->Parisian).Val.V_BOOL == TRUE)
        return OK;


  return  WRONG;
}



static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  static int first = 1;

  if (first)
    {
      Met->Par[0].Val.V_PDOUBLE = 0.01;
      Met->Par[1].Val.V_PINT = 15;
      Met->Par[2].Val.V_PINT = 15;
      first = 0;
    }


  return OK;
}



PricingMethod MET(AP_LaplaceDoubleParisian) =
{
  "AP_Laplace_Double_Parisian",
  { {"Delta Increment Rel", PDOUBLE, {100}, ALLOW},
    {"sum truncation", PINT, {15}, ALLOW},
    {"window average", PINT, {15}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(AP_LaplaceDoubleParisian),
  { {"Price", DOUBLE, {100}, FORBID},
    {"Delta", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(AP_LaplaceDoubleParisian),
```

```
    CHK_ok,
    MET(Init)
  } ;
}
```