Help

```
#include <stdlib.h>
#include  "
href../../../mod/dup1d/dup1d_std/dup1d_std_h_src.pdfdup1d_std.h"

#define DELTA_MAX_ADAPT_DUP 0.2
#define EPS_MAX_ADAPT_DUP 0.2

#define NA_MAX_ADAPT_DUP 1000
#define M_MAX_ADAPT_DUP 1000
#define TOL_ADAPT_DUP 0.001


int sigma_type;
static double **Primal_Price = NULL, * *Primal_Price0 = NULL, * *Primal_Price1 =
static double *Price_Coarse = NULL, *Price_Fine = NULL, *x_Coarse = NULL, *Proj
static int **x_hier = NULL, * *new_x_hier = NULL, *times_hier = NULL, *new_times

static  double a_ind = 0., b_ind = 0.;
static double a_norm2 = 0., x_min = 0., x_max = 0.;
static double global_times_error = 0., global_space_error = 0.;
static int NA_ADAPT = 0, NA_Coarse = 0, n_element = 0;
static int *NA = NULL, *new_NA = NULL, *NA_Coarse1 = NULL;
static int M = 0, new_M = 0, M_Coarse = 0;


/* EDP Discretization terms */
static double diff(double t, double r, double divid, double x, double y)
{
  return 0.5 * (SQR(x * premia_local_vol(t, x, sigma_type)) + SQR(y * premia_loc
}

static double conv(double t, double r, double divid, double x, double y, double
{
  return (w1 * (SQR(x) * premia_local_vol_x(t, x, sigma_type) * premia_local_vol
}

static double conv_dual(double t, double r, double divid, double x, double y, do
{
```

```
  return (w1 * (x * (SQR(premia_local_vol(t, x, sigma_type)) - (r - divid)) + SQ
}

static double source1(double t, double r, double divid, double x, double y, doub
{
  return (w1 * (((r - divid) + r - SQR(premia_local_vol(t, x, sigma_type)) - SQR
}

static double source2(double t, double r, double divid, double x, double y, doub
{
  return w1 + w2;
}

/* Memory allocation */
static void memory_allocation()
{
  int i;

  times = malloc((M_MAX_ADAPT_DUP + 1) * sizeof(double));
  times_Coarse1 = malloc((M_MAX_ADAPT_DUP + 1) * sizeof(double));
  times_hier = (int *)calloc(M_MAX_ADAPT_DUP + 1, sizeof(int));
  new_times_hier = (int *)calloc(M_MAX_ADAPT_DUP + 1, sizeof(int));
  new_times = malloc((M_MAX_ADAPT_DUP + 1) * sizeof(double));
  times_error = malloc((M_MAX_ADAPT_DUP + 1) * sizeof(double));

  NA = malloc((M_MAX_ADAPT_DUP + 1) * sizeof(int));
  NA_Coarse1 = malloc((M_MAX_ADAPT_DUP + 1) * sizeof(int));
  new_NA = malloc((M_MAX_ADAPT_DUP + 1) * sizeof(int));

  Price_Coarse = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));
  x_Coarse = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));
  Price_Fine = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));
  v_error = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));
  v_error_Coarse = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));
  Proj = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));

  x = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
  for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
    x[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

  x_Coarse1 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
```

```
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  x_Coarse1[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

x_hier = (int **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(int *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  x_hier[i] = (int *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(int));

new_x_hier = (int **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(int *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  new_x_hier[i] = (int *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(int));

new_x = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  new_x[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Primal_Price = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Primal_Price[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Price_Coarse1 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Price_Coarse1[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Price_Coarse0 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Price_Coarse0[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Primal_Price0 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Primal_Price0[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Primal_Price1 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Primal_Price1[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Primal_Proj = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Primal_Proj[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Primal_Proj0 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
```

```
  Primal_Proj0[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Primal_Proj1 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Primal_Proj1[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Dual_Price = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Dual_Price[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Dual_Price0 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Dual_Price0[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Dual_Price1 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Dual_Price1[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Dual_Proj = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Dual_Proj[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Dual_Proj1 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Dual_Proj1[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Dual_Proj2 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Dual_Proj2[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

Error = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  Error[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

v_error1 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  v_error1[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

v_error_Coarse0 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  v_error_Coarse0[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));
```

```
  v_error_Coarse1 = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
  for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
    v_error_Coarse1[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

  space_error = (double **)calloc(M_MAX_ADAPT_DUP + 1, sizeof(double *));
  for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
    space_error[i] = (double *)calloc(NA_MAX_ADAPT_DUP + 1, sizeof(double));

  return;
}


/*Memory Desallocation*/
static void free_memory()
{
  int i;

  free(NA);
  free(NA_Coarse1);
  free(new_NA);

  free(times);
  free(times_Coarse1);
  free(times_hier);
  free(new_times_hier);
  free(times_error);
  free(new_times);

  free(v_error);
  free(v_error_Coarse);
  free(Price_Coarse);
  free(x_Coarse);
  free(Price_Fine);
  free(Proj);

  for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
    free(x[i]);
  free(x);

  for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
```

```
  free(new_x[i]);
free(new_x);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(x_Coarse1[i]);
free(x_Coarse1);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(x_hier[i]);
free(x_hier);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(new_x_hier[i]);
free(new_x_hier);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Primal_Price[i]);
free(Primal_Price);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Primal_Price0[i]);
free(Primal_Price0);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Primal_Price1[i]);
free(Primal_Price1);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Price_Coarse0[i]);
free(Price_Coarse0);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Price_Coarse1[i]);
free(Price_Coarse1);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Primal_Proj[i]);
free(Primal_Proj);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Primal_Proj0[i]);
```

```
free(Primal_Proj0);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Primal_Proj1[i]);
free(Primal_Proj1);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Dual_Price[i]);
free(Dual_Price);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Dual_Price0[i]);
free(Dual_Price0);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Dual_Price1[i]);
free(Dual_Price1);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Dual_Proj[i]);
free(Dual_Proj);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Dual_Proj1[i]);
free(Dual_Proj1);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Dual_Proj2[i]);
free(Dual_Proj2);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(Error[i]);
free(Error);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(v_error1[i]);
free(v_error1);

for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
  free(v_error_Coarse1[i]);
free(v_error_Coarse1);
```

```
  for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
    free(v_error_Coarse0[i]);
  free(v_error_Coarse0);

  for (i = 0; i < M_MAX_ADAPT_DUP + 1; i++)
    free(space_error[i]);
  free(space_error);

  return;
}

/*Primal Solver for first rappresentative error*/
static int primal_solver0(NumFunc_1  *p, double s, double t, double r, double di
{
  double omega, epsilon, K;

  double b, c, a1, a2, a3, a4, error, y, norm;
  int i, j, loops, TimesIndex;
  double *P_Old, *Obst, *Rhs, *alpha4, *beta4, *gamma4, *alpha1, *beta1, *gamma1
  double hi, hip, w1, w2;
  double eta0i, eta1i, eta0ip, eta1ip, diffi, convi, diffip, convip;


  /*Memory Allocation*/
  alpha1 = malloc((2 * NA0 + 2) * sizeof(double));
  beta1 = malloc((2 * NA0 + 2) * sizeof(double));
  gamma1 = malloc((2 * NA0 + 2) * sizeof(double));
  alpha2 = malloc((2 * NA0 + 2) * sizeof(double));
  beta2 = malloc((2 * NA0 + 2) * sizeof(double));
  gamma2 = malloc((2 * NA0 + 2) * sizeof(double));
  alpha3 = malloc((2 * NA0 + 2) * sizeof(double));
  beta3 = malloc((2 * NA0 + 2) * sizeof(double));
  gamma3 = malloc((2 * NA0 + 2) * sizeof(double));
  alpha4 = malloc((2 * NA0 + 2) * sizeof(double));
  beta4 = malloc((2 * NA0 + 2) * sizeof(double));
  gamma4 = malloc((2 * NA0 + 2) * sizeof(double));
  alpha5 = malloc((2 * NA0 + 2) * sizeof(double));
  beta5 = malloc((2 * NA0 + 2) * sizeof(double));
  gamma5 = malloc((2 * NA0 + 2) * sizeof(double));
  vec_tme = malloc((M0 + 1) * sizeof(double));
```

```
news_x = malloc((2 * NA0 + 2) * sizeof(double));
P_Old = malloc((2 * NA0 + 2) * sizeof(double));
Obst = malloc((2 * NA0 + 2) * sizeof(double));
Rhs = malloc((2 * NA0 + 2) * sizeof(double));

omega = 1.5;
epsilon = 1.0e-9;
K = p->Par[0].Val.V_DOUBLE;
/*Space Localisation*/

for (i = 0; i <= M0; i++) vec_tme[i] = ((double)i) * (t) / (double)M0;

for (i = 0; i <= NA0; i++)
  {

    news_x[i] = x_min + ((double)i) * (x_max - x_min) / (double)NA0;
    P_Old[i] = (p->Compute)(p->Par, news_x[i]);
    Obst[i] = P_Old[i];

    P_Old[i + NA0] = 0.;

  }
P_Old[2 * NA0 + 1] = 0.;
/*Finite Difference Cycle*/

for (TimesIndex = 1; TimesIndex <= M0; TimesIndex++)
  {

    a1 = (1. + r * (vec_tme[TimesIndex] - vec_tme[TimesIndex - 1])) / 6.;

    b = (vec_tme[TimesIndex] - vec_tme[TimesIndex - 1]) / 2.;

    c = (vec_tme[TimesIndex] - vec_tme[TimesIndex - 1]) / 2.;

    a2 = (1. + 0.5 * r * (vec_tme[TimesIndex] - vec_tme[TimesIndex - 1])) / 6.

    a3 = (0.5 * r * (vec_tme[TimesIndex] - vec_tme[TimesIndex - 1])) / 6.;

    a4 = (1. / 2. + 1. / 3.*r * (vec_tme[TimesIndex] - vec_tme[TimesIndex - 1]

    /*Computation of Lhs coefficients*/
```

```
for (i = 1; i < NA0; i++)

  {
    hi = news_x[i] - news_x[i - 1];
    hip = news_x[i + 1] - news_x[i];
    eta0i = 0.5 * (news_x[i] + news_x[i - 1]) - sqrt(3.) / 6.*(news_x[i] -
    eta1i = 0.5 * (news_x[i] + news_x[i - 1]) + sqrt(3.) / 6.*(news_x[i] -
    eta0ip = 0.5 * (news_x[i + 1] + news_x[i]) - sqrt(3.) / 6.*(news_x[i +
    eta1ip = 0.5 * (news_x[i + 1] + news_x[i]) + sqrt(3.) / 6.*(news_x[i +

    diffi = diff(vec_tme[TimesIndex], r, divid, eta0i, eta1i) / hi;
    diffip = diff(vec_tme[TimesIndex], r, divid, eta0ip, eta1ip) / hip;
    w1 = (eta0i - news_x[i - 1]) / hi;
    w2 = (eta1i - news_x[i - 1]) / hi;
    convi = conv(vec_tme[TimesIndex], r, divid, eta0i, eta1i, w1, w2);
    w1 = (news_x[i + 1] - eta0ip) / hip;
    w2 = (news_x[i + 1] - eta1ip) / hip;
    convip = conv(vec_tme[TimesIndex], r, divid, eta0ip, eta1ip, w1, w2);

    alpha1[i] = a1 * hi - b * diffi - c * convi;
    beta1[i] = 2.*a1 * (hi + hip) +
               b * (diffi + diffip) + c * (convi - convip);
    gamma1[i] = a1 * hip - b * diffip + c * convip;

    alpha2[i] = a2 * hi - 0.5 * b * diffi - 0.5 * c * convi;
    beta2[i] = 2.*a2 * (hi + hip) +
               0.5 * b * (diffi + diffip) + 0.5 * c * (convi - convip);
    gamma2[i] = a2 * hip - 0.5 * b * diffip + 0.5 * c * convip;

    alpha3[i] = a3 * hi - 0.5 * b * diffi - 0.5 * c * convi;
    beta3[i] = 2.*a3 * (hi + hip) +
               0.5 * b * (diffi + diffip) + 0.5 * c * (convi - convip);
    gamma3[i] = a3 * hip - 0.5 * b * diffip + 0.5 * c * convip;

    alpha4[i] = a4 * hi - (1. / 3.) * b * diffi - (1. / 3.) * c * convi;
    beta4[i] = 2.*a4 * (hi + hip) +
               (1. / 3.) * b * (diffi + diffip) + (1. / 3.) * c * (convi -
    gamma4[i] = a4 * hip - (1. / 3.) * b * diffip + (1. / 3.) * c * convip

    /*Rhs*/
```

```
    alpha5[i] = hi / 6.;
    beta5[i] = (hi + hip) / 3.;
    gamma5[i] = hip / 6.;
  }

/*Init Rhs*/
for (j = 1; j <= NA0 - 1; j++)
  Rhs[j] = alpha5[j] * (P_Old[j - 1] + P_Old[NA0 + j]) + beta5[j] * (P_Old

for (j = NA0 + 2; j <= 2 * NA0; j++) Rhs[j] = 0.;

if ((p->Compute) == &Call)
  {
    P_Old[0] = 0.;
    P_Old[NA0] = x_max * exp(-divid * vec_tme[TimesIndex - 1]) - K * exp(-
    P_Old[NA0 + 1] = 0.;
    P_Old[2 * NA0 + 1] = x_max * exp(-divid * vec_tme[TimesIndex]) - K * e
  }
else if ((p->Compute) == &Put)
  {
    P_Old[0] = K * exp(-r * vec_tme[TimesIndex - 1]) - x_min * exp(-divid
    P_Old[NA0] = 0.;
    P_Old[NA0 + 1] = K * exp(-r * vec_tme[TimesIndex]) - x_min * exp(-divi
    P_Old[2 * NA0 + 1] = 0.;
  }

/*Psor Cycle*/
loops = 0;
do
  {
    error = 0.;

    norm = 0.;

    for (j = 1; j <= NA0 - 1; j++)
      {
        y = (Rhs[j] - alpha1[j] * P_Old[j - 1] - gamma1[j] * P_Old[j + 1]

        y = P_Old[j] + omega * (y - P_Old[j]);

        error += (double)(j + 1) * fabs(y - P_Old[j]);
```

```
            norm += fabs(y);

            P_Old[j] = y;
          }

        for (j = NA0 + 2; j <= 2 * NA0; j++)
          {
            y = (Rhs[j] - alpha4[j - NA0 - 1] * P_Old[j - 1] - gamma4[j - NA0


            y = P_Old[j] + omega * (y - P_Old[j]);

            error += (double)(j + 1) * fabs(y - P_Old[j]);

            norm += fabs(y);

            P_Old[j] = y;

          }

        if (norm < 1.0) norm = 1.0;

        error = error / norm;
        loops++;

      }
    while ((error > epsilon) && (loops < MAXLOOPS));

    /*End Psor Cycle*/
    for (i = 0; i <= NA0; i++)
      {
        Price_Coarse0[TimesIndex][i] = P_Old[i];
        Price_Coarse1[TimesIndex][i] = P_Old[i + NA0 + 1];
      }
  }

/*End Finite Difference Cycle*/
for (i = 0; i <= NA0; i++)
  {
    Primal0[i] = P_Old[i] + P_Old[i + NA0 + 1];
```

```
    }


  /*Memory Desallocation*/

  free(P_Old);
  free(Obst);
  free(Rhs);
  free(alpha2);
  free(beta2);
  free(gamma2);
  free(alpha1);
  free(beta1);
  free(gamma1);
  free(alpha3);
  free(beta3);
  free(gamma3);
  free(alpha4);
  free(beta4);
  free(gamma4);
  free(alpha5);
  free(beta5);
  free(gamma5);
  free(vec_tme);
  free(news_x);

  return 0;

}

/* FEM Discontinous Galerkin Method q=1 for solve Primal Problem */
static int primal_solver(NumFunc_1  *p, double s, double t, double r, double div
{
  double omega, epsilon, K;
  int TimesIndex;
  double b, c, a1, a2, a3, a4, error, y, norm;
  int i, j, loops, NAloc;
  double *P_Old, *P_Proj0, *P_Proj1, *Obst, *Rhs, *alpha4, *beta4, *gamma4, *alp
  double hi, hip, w1, w2;
  double eta0i, eta1i, eta0ip, eta1ip, diffi, convi, diffip, convip;
```

```
/*Memory Allocation*/
alpha1 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
beta1 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
gamma1 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
alpha2 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
beta2 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
gamma2 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
alpha3 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
beta3 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
gamma3 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
alpha4 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
beta4 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
gamma4 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
alpha5 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
beta5 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
gamma5 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
P_Old = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
P_Proj0 = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));
P_Proj1 = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));
Obst = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
Rhs = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));

omega = 1.5;
epsilon = 1.e-9;
K = p->Par[0].Val.V_DOUBLE;

/*Space Localisation*/
NAloc = NA[0];
for (i = 0; i <= NAloc; i++)
  {
    P_Old[i] = (p->Compute)(p->Par, x[0][i]);
    Obst[i] = P_Old[i];
    P_Old[i + NAloc] = 0.;
  }
P_Old[2 * NAloc + 1] = 0.;
for (i = 0; i <= NAloc; i++)
  Primal_Price[0][i] = P_Old[i];

/*Finite Difference Cycle*/
```

```
for (TimesIndex = 1; TimesIndex <= M; TimesIndex++)
  {
    NAloc = NA[TimesIndex];

    /* Projection on Cin */
    for (i = 0; i <= NA[TimesIndex - 1]; i++)
      P_Proj0[i] = P_Old[i];

    for (i = 0; i <= NA[TimesIndex - 1]; i++)
      P_Proj1[i] = P_Old[i + NA[TimesIndex - 1] + 1];

    for (i = 1; i < NA[TimesIndex]; i++)
      {
        j = 1;
        while (x[TimesIndex - 1][j] < x[TimesIndex][i]) j++;
        P_Old[i] = P_Proj0[j - 1] * (x[TimesIndex - 1][j] - x[TimesIndex][i])
        P_Old[i + NAloc + 1] = P_Proj1[j - 1] * (x[TimesIndex - 1][j] - x[Time
      }

    if ((p->Compute) == &Call)
      {
        P_Old[0] = 0.;
        P_Old[NAloc] = P_Proj0[NA[TimesIndex - 1]];
        P_Old[NAloc + 1] = 0.;
        P_Old[2 * NAloc + 1] = P_Proj1[NA[TimesIndex - 1]];
      }
    else if ((p->Compute) == &Put)
      {
        P_Old[0] = P_Proj0[0];
        P_Old[NAloc] = 0.;
        P_Old[NAloc + 1] = P_Proj1[0];
        P_Old[2 * NAloc + 1] = 0.;
      }

    Primal_Proj[TimesIndex][0] = P_Proj0[0] + P_Proj1[0];
    Primal_Proj[TimesIndex][NAloc] = P_Proj0[NA[TimesIndex - 1]] + P_Proj1[NA[
    Primal_Proj0[TimesIndex][0] = P_Proj0[0];
    Primal_Proj0[TimesIndex][NAloc] = P_Proj0[NA[TimesIndex - 1]];
    Primal_Proj1[TimesIndex][0] = P_Proj1[0];
    Primal_Proj1[TimesIndex][NAloc] = P_Proj1[NA[TimesIndex - 1]];
```

```
for (i = 1; i < NAloc; i++)
  {
    Primal_Proj[TimesIndex][i] = P_Old[i] + P_Old[i + NAloc + 1];

    Primal_Proj0[TimesIndex][i] = P_Old[i];
    Primal_Proj1[TimesIndex][i] = P_Old[i + NAloc + 1];
  }

a1 = (1. + r * (times[TimesIndex] - times[TimesIndex - 1])) / 6.;

b = (times[TimesIndex] - times[TimesIndex - 1]) / 2.;

c = (times[TimesIndex] - times[TimesIndex - 1]) / 2.;

a2 = (1. + 0.5 * r * (times[TimesIndex] - times[TimesIndex - 1])) / 6.;

a3 = (0.5 * r * (times[TimesIndex] - times[TimesIndex - 1])) / 6.;

a4 = (1. / 2. + 1. / 3.*r * (times[TimesIndex] - times[TimesIndex - 1])) /
/*Computation of Lhs coefficients */
for (i = 1; i < NAloc; i++)
  {
    hi = x[TimesIndex][i] - x[TimesIndex][i - 1];
    hip = x[TimesIndex][i + 1] - x[TimesIndex][i];
    eta0i = 0.5 * (x[TimesIndex][i] + x[TimesIndex][i - 1]) - sqrt(3.) / 6
    eta1i = 0.5 * (x[TimesIndex][i] + x[TimesIndex][i - 1]) + sqrt(3.) / 6
    eta0ip = 0.5 * (x[TimesIndex][i + 1] + x[TimesIndex][i]) - sqrt(3.) /
    eta1ip = 0.5 * (x[TimesIndex][i + 1] + x[TimesIndex][i]) + sqrt(3.) /
    diffi = diff(times[TimesIndex], r, divid, eta0i, eta1i) / hi;
    diffip = diff(times[TimesIndex], r, divid, eta0ip, eta1ip) / hip;
    w1 = (eta0i - x[TimesIndex][i - 1]) / hi;
    w2 = (eta1i - x[TimesIndex][i - 1]) / hi;
    convi = conv(times[TimesIndex], r, divid, eta0i, eta1i, w1, w2);
    w1 = (x[TimesIndex][i + 1] - eta0ip) / hip;
    w2 = (x[TimesIndex][i + 1] - eta1ip) / hip;
    convip = conv(times[TimesIndex], r, divid, eta0ip, eta1ip, w1, w2);

    alpha1[i] = a1 * hi - b * diffi - c * convi;
    beta1[i] = 2.*a1 * (hi + hip) +
               b * (diffi + diffip) + c * (convi - convip);
    gamma1[i] = a1 * hip - b * diffip + c * convip;
```

```
    alpha2[i] = a2 * hi - 0.5 * b * diffi - 0.5 * c * convi;
    beta2[i] = 2.*a2 * (hi + hip) +
               0.5 * b * (diffi + diffip) + 0.5 * c * (convi - convip);
    gamma2[i] = a2 * hip - 0.5 * b * diffip + 0.5 * c * convip;

    alpha3[i] = a3 * hi - 0.5 * b * diffi - 0.5 * c * convi;
    beta3[i] = 2.*a3 * (hi + hip) +
               0.5 * b * (diffi + diffip) + 0.5 * c * (convi - convip);
    gamma3[i] = a3 * hip - 0.5 * b * diffip + 0.5 * c * convip;

    alpha4[i] = a4 * hi - (1. / 3.) * b * diffi - (1. / 3.) * c * convi;
    beta4[i] = 2.*a4 * (hi + hip) +
               (1. / 3.) * b * (diffi + diffip) + (1. / 3.) * c * (convi -
    gamma4[i] = a4 * hip - (1. / 3.) * b * diffip + (1. / 3.) * c * convip

    /*Rhs*/
    alpha5[i] = hi / 6.;
    beta5[i] = (hi + hip) / 3.;
    gamma5[i] = hip / 6.;
  }

/*Init Rhs*/
for (j = 1; j <= NAloc - 1; j++)
  Rhs[j] = alpha5[j] * (P_Old[j - 1] + P_Old[NAloc + j]) + beta5[j] * (P_O

for (j = NAloc + 2; j <= 2 * NAloc; j++) Rhs[j] = 0.;

if ((p->Compute) == &Call)
  {
    P_Old[0] = 0.;
    P_Old[NAloc] = x[TimesIndex][NAloc] * exp(-divid * (times[TimesIndex -
    P_Old[NAloc + 1] = 0.;
    P_Old[2 * NAloc + 1] = x[TimesIndex][NAloc] * exp(-divid * (times[Time
  }
else if ((p->Compute) == &Put)
  {
    P_Old[0] = K * exp(-r * times[TimesIndex - 1]) - x[TimesIndex][0] * ex
    P_Old[NAloc] = 0.;
    P_Old[NAloc + 1] = K * exp(-r * times[TimesIndex]) - x[TimesIndex][0]
    P_Old[2 * NAloc + 1] = 0.;
```

```
  }

/*Psor Cycle*/
loops = 0;
do
  {

    error = 0.;

    norm = 0.;

    for (j = 1; j <= NAloc - 1; j++)
      {
        y = (Rhs[j] - alpha1[j] * P_Old[j - 1] - gamma1[j] * P_Old[j + 1]


        y = P_Old[j] + omega * (y - P_Old[j]);

        error += (double)(j + 1) * fabs(y - P_Old[j]);

        norm += fabs(y);

        P_Old[j] = y;
      }

    for (j = NAloc + 2; j <= 2 * NAloc; j++)
      {
        y = (Rhs[j] - alpha4[j - NAloc - 1] * P_Old[j - 1] - gamma4[j - NA


        y = P_Old[j] + omega * (y - P_Old[j]);

        error += (double)(j + 1) * fabs(y - P_Old[j]);

        norm += fabs(y);

        P_Old[j] = y;

      }

    if (norm < 1.0) norm = 1.0;
```

```
        error = error / norm;

        loops++;


      }
    while ((error > epsilon) && (loops < MAXLOOPS));

    /*End Psor Cycle*/
    for (i = 0; i <= NAloc; i++)
      {
        Primal_Price[TimesIndex][i] = P_Old[i] + P_Old[i + NAloc + 1];
        Primal_Price0[TimesIndex][i] = P_Old[i];
        Primal_Price1[TimesIndex][i] = P_Old[i + NAloc + 1];
        /*Primal_Proj[TimesIndex][i]=P_NAew[i];*/
      }
  }

/*End Finite Difference Cycle*/

/*Memory Desallocation*/

free(P_Old);
free(P_Proj0);
free(P_Proj1);
free(Obst);
free(Rhs);
free(alpha2);
free(beta2);
free(gamma2);
free(alpha1);
free(beta1);
free(gamma1);
free(alpha3);
free(beta3);
free(gamma3);
free(alpha4);
free(beta4);
free(gamma4);
free(alpha5);
```

```
  free(beta5);
  free(gamma5);

  return 0;


}

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

/*                          Dual Problem                          */

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

/* FEM Discontinous Galerkin Method q=1 for solve Dual Problem */
static int dual_solver(double s, double t, double r, double divid)
{
  double omega, epsilon;

  double error, y, norm;
  int i, j, loops, TimesIndex, NAloc;
  double *P_Old, *Obst, *Rhs, *P_Proj0, *P_Proj1, *alpha4, *beta4, *gamma4, *alp
  double hi, hip, w1, w2;
  double eta0i, eta1i, eta0ip, eta1ip, diffi, convi, diffip, convip;

  double b, c, a1, a2, a3, a4, a1m, a2m, a3m, a4m, a1p, a2p, a3p, a4p;
  double sourcei1, sourceim1, sourceip1, sourcei2, sourceim2, sourceip2;



  /*Memory Allocation*/

  alpha1 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
  beta1 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
  gamma1 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
  alpha2 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
  beta2 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
  gamma2 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
  alpha3 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
  beta3 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
  gamma3 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
```

```
alpha4 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
beta4 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
gamma4 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
alpha5 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
beta5 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
gamma5 = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
P_Old = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
P_Proj0 = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));
P_Proj1 = malloc((NA_MAX_ADAPT_DUP + 1) * sizeof(double));
Obst = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));
Rhs = malloc((NA_MAX_ADAPT_DUP) * sizeof(double));

omega = 1.5;
epsilon = 1.0e-9;

NAloc = NA[M];
for (i = 0; i <= NAloc; i++)
  {
    P_Old[i] = v_error[i];
    P_Old[i + NAloc] = 0.;
  }
P_Old[2 * NAloc + 1] = 0.;
NA[M + 1] = NA[M];
for (i = 0; i <= NAloc; i++)
  {
    Dual_Price[M + 1][i] = P_Old[i];
    x[M + 1][i] = x[M][i];
  }

/*Finite Difference Cycle*/

for (TimesIndex = M; TimesIndex >= 1; TimesIndex--)
  {
    NAloc = NA[TimesIndex];
    P_Old[0] = 0.;
    P_Old[NAloc] = 0.;
    P_Old[NAloc + 1] = 0.;
    P_Old[2 * NAloc + 1] = 0.;

    /* Projection on Cin */
    for (i = 0; i <= NA[TimesIndex + 1]; i++)
```

```
  P_Proj0[i] = P_Old[i];

for (i = 0; i <= NA[TimesIndex + 1]; i++)
  P_Proj1[i] = P_Old[i + NA[TimesIndex + 1] + 1];

for (i = 1; i < NA[TimesIndex]; i++)
  {
    j = 1;
    while (x[TimesIndex + 1][j] < x[TimesIndex][i]) j++;
    P_Old[i] = P_Proj0[j - 1] * (x[TimesIndex + 1][j] - x[TimesIndex][i])

    P_Old[i + NAloc + 1] = P_Proj1[j - 1] * (x[TimesIndex + 1][j] - x[Time
  }

for (i = 1; i < NAloc; i++)
  {
    Dual_Proj[TimesIndex][i] = P_Old[i];
    Dual_Proj1[TimesIndex][i] = P_Old[i + NAloc + 1];
  }

Dual_Proj[TimesIndex][0] = 0.;
Dual_Proj[TimesIndex][NAloc] = 0.;

Dual_Proj1[TimesIndex][0] = 0.;
Dual_Proj1[TimesIndex][NAloc] = 0.;

b = (times[TimesIndex] - times[TimesIndex - 1]) / 2.;
c = (times[TimesIndex] - times[TimesIndex - 1]) / 2.;

/*Computation of Lhs coefficients*/
for (i = 1; i < NAloc; i++)
  {
    hi = x[TimesIndex][i] - x[TimesIndex][i - 1];
    hip = x[TimesIndex][i + 1] - x[TimesIndex][i];

    eta0i = 0.5 * (x[TimesIndex][i] + x[TimesIndex][i - 1]) - sqrt(3.) / 6
    eta1i = 0.5 * (x[TimesIndex][i] + x[TimesIndex][i - 1]) + sqrt(3.) / 6
    eta0ip = 0.5 * (x[TimesIndex][i + 1] + x[TimesIndex][i]) - sqrt(3.) /
    eta1ip = 0.5 * (x[TimesIndex][i + 1] + x[TimesIndex][i]) + sqrt(3.) /
    diffi = diff(times[TimesIndex], r, divid, eta0i, eta1i) / hi;
    diffip = diff(times[TimesIndex], r, divid, eta0ip, eta1ip) / hip;
```

```
w1 = (eta0i - x[TimesIndex][i - 1]) / hi;
w2 = (eta1i - x[TimesIndex][i - 1]) / hi;
convi = conv_dual(times[TimesIndex], r, divid, eta0i, eta1i, w1, w2);
w1 = (x[TimesIndex][i + 1] - eta0ip) / hip;
w2 = (x[TimesIndex][i + 1] - eta1ip) / hip;
convip = conv_dual(times[TimesIndex], r, divid, eta0ip, eta1ip, w1, w2


w1 = SQR(eta0i - x[TimesIndex][i - 1]) / hi;
w2 = SQR(eta1i - x[TimesIndex][i - 1]) / hi;
sourcei1 = source1(times[TimesIndex], r, divid, eta0i, eta1i, w1, w2)
sourcei2 = source2(times[TimesIndex], r, divid, eta0i, eta1i, w1, w2)

w1 = SQR(x[TimesIndex][i + 1] - eta0ip) / hip;
w2 = SQR(x[TimesIndex][i + 1] - eta1ip) / hip;
sourcei1 += source1(times[TimesIndex], r, divid, eta0ip, eta1ip, w1, w
sourcei2 += source2(times[TimesIndex], r, divid, eta0ip, eta1ip, w1, w


w1 = (x[TimesIndex][i] - eta0i) * (eta0i - x[TimesIndex][i - 1]) / hi;
w2 = (x[TimesIndex][i] - eta1i) * (eta1i - x[TimesIndex][i - 1]) / hi;
sourceim1 = source1(times[TimesIndex], r, divid, eta0i, eta1i, w1, w2)
sourceim2 = source2(times[TimesIndex], r, divid, eta0i, eta1i, w1, w2)

w1 = (x[TimesIndex][i + 1] - eta0ip) * (eta0ip - x[TimesIndex][i]) / h
w2 = (x[TimesIndex][i + 1] - eta1ip) * (eta1ip - x[TimesIndex][i]) / h
sourceip1 = source1(times[TimesIndex], r, divid, eta0ip, eta1ip, w1, w
sourceip2 = source2(times[TimesIndex], r, divid, eta0ip, eta1ip, w1, w

a1 = sourcei2 + sourcei1 * (times[TimesIndex] - times[TimesIndex - 1])

a1p = sourceip2 + sourceip1 * (times[TimesIndex] - times[TimesIndex -
a1m = sourceim2 + sourceim1 * (times[TimesIndex] - times[TimesIndex -

a2 = sourcei2 + 0.5 * sourcei1 * (times[TimesIndex] - times[TimesIndex
a2p = sourceip2 + 0.5 * sourceip1 * (times[TimesIndex] - times[TimesIn
a2m = sourceim2 + 0.5 * sourceim1 * (times[TimesIndex] - times[TimesIn

a3 = 0.5 * sourcei1 * (times[TimesIndex] - times[TimesIndex - 1]);
a3p = 0.5 * sourceip1 * (times[TimesIndex] - times[TimesIndex - 1]);
a3m = 0.5 * sourceim1 * (times[TimesIndex] - times[TimesIndex - 1]);
```

```
    a4 = 1. / 2.*sourcei2 + 1. / 3.*sourcei1 * (times[TimesIndex] - times[
    a4p = 1. / 2.*sourceip2 + 1. / 3.*sourceip1 * (times[TimesIndex] - tim
    a4m = 1. / 2.*sourceim2 + 1. / 3.*sourceim1 * (times[TimesIndex] - tim

    alpha1[i] = a1m - b * diffi - c * convi;

    beta1[i] = a1 +
               b * (diffi + diffip) + c * (convi - convip);
    gamma1[i] = a1p - b * diffip + c * convip;

    alpha2[i] = a2m - 0.5 * b * diffi - 0.5 * c * convi;
    beta2[i] = a2 +
               0.5 * b * (diffi + diffip) + 0.5 * c * (convi - convip);
    gamma2[i] = a2p - 0.5 * b * diffip + 0.5 * c * convip;

    alpha3[i] = a3m - 0.5 * b * diffi - 0.5 * c * convi;
    beta3[i] = a3 +
               0.5 * b * (diffi + diffip) + 0.5 * c * (convi - convip);
    gamma3[i] = a3p - 0.5 * b * diffip + 0.5 * c * convip;

    alpha4[i] = a4m - (1. / 3.) * b * diffi - (1. / 3.) * c * convi;
    beta4[i] = a4 +
               (1. / 3.) * b * (diffi + diffip) + (1. / 3.) * c * (convi -
    gamma4[i] = a4p - (1. / 3.) * b * diffip + (1. / 3.) * c * convip;

    /*Rhs*/
    alpha5[i] = hi / 6.;
    beta5[i] = (hi + hip) / 3.;
    gamma5[i] = hip / 6.;

  }

/*Init Rhs*/
/*Init Rhs*/
for (j = 1; j < NAloc; j++)
  Rhs[j] = alpha5[j] * P_Old[j - 1] + beta5[j] * P_Old[j] + gamma5[j] * P_

for (j = NAloc + 2; j <= 2 * NAloc; j++) Rhs[j] = 0.;

/*Psor Cycle*/
```

```
loops = 0;
do
  {
    error = 0.;

    norm = 0.;

    for (j = 1; j < NAloc; j++)

      {
        y = (Rhs[j] - alpha1[j] * P_Old[j - 1] - gamma1[j] * P_Old[j + 1]


        y = P_Old[j] + omega * (y - P_Old[j]);

        error += (double)(j + 1) * fabs(y - P_Old[j]);

        norm += fabs(y);

        P_Old[j] = y;

      }

    for (j = NAloc + 2; j <= 2 * NAloc; j++)
      {
        y = (Rhs[j] - alpha4[j - NAloc - 1] * P_Old[j - 1] - gamma4[j - NA


        y = P_Old[j] + omega * (y - P_Old[j]);

        error += (double)(j + 1) * fabs(y - P_Old[j]);

        norm += fabs(y);

        P_Old[j] = y;
      }

    if (norm < 1.0) norm = 1.0;

    error = error / norm;
```

```
        loops++;
      }

    while ((error > epsilon) && (loops < MAXLOOPS));

    /*End Psor Cycle*/
    P_Old[0] = 0.;
    P_Old[NAloc] = 0.;
    P_Old[NAloc + 1] = 0.;
    P_Old[2 * NAloc + 1] = 0.;

    for (i = 0; i <= NAloc; i++)
      {
        Dual_Price[TimesIndex][i] = P_Old[i] + P_Old[i + NAloc + 1];
        Dual_Price0[TimesIndex][i] = P_Old[i];
        Dual_Price1[TimesIndex][i] = P_Old[i + NAloc + 1];
      }

  }
/*End Finite Difference Cycle*/
NA[0] = NA[1];
for (i = 0; i <= NAloc; i++)
  {
    Dual_Price[0][i] = Dual_Price[1][i];
    x[0][i] = x[1][i];
  }

for (TimesIndex = 2; TimesIndex <= M; TimesIndex++)
  {
    for (i = 1; i < NA[TimesIndex]; i++)
      {
        j = 1;
        while (x[TimesIndex - 1][j] < x[TimesIndex][i]) j++;
        Dual_Proj2[TimesIndex][i] = Dual_Price1[TimesIndex - 1][j - 1] * (x[Ti

      }
    Dual_Proj2[TimesIndex][0] = 0.;
    Dual_Proj2[TimesIndex][NA[TimesIndex]] = 0.;
  }
/*Memory Desallocation*/
```

```
  free(P_Old);
  free(P_Proj0);
  free(P_Proj1);
  free(Obst);
  free(Rhs);
  free(alpha2);
  free(beta2);
  free(gamma2);
  free(alpha1);
  free(beta1);
  free(gamma1);
  free(alpha3);
  free(beta3);
  free(gamma3);
  free(alpha4);
  free(beta4);
  free(gamma4);
  free(alpha5);
  free(beta5);

  free(gamma5);

  return 0;
}



/*******************************************************/
/*              ADAPTIVE PROCEDURES                    */
/*******************************************************/

/* Space Refinement */
static int space_refine(int j, int i, int numb)
{
  new_x[j][numb] = x[j][i - 1] + 0.5 * (x[j][i] - x[j][i - 1]);
  new_x[j][numb + 1] = x[j][i];
  new_x_hier[j][numb] = x_hier[j][i];
  new_x_hier[j][numb + 1] = x_hier[j][i];

  return 1;
```

```
}

/* Space DeRefinement */
static int space_derefine(int j, int i, int numb)
{
  new_x[j][numb] = x[j][i + 1];
  new_x_hier[j][numb] = x_hier[j][i];

  return 1;
}
/* Space OK */
static int space_ok(int j, int i, int numb)
{
  new_x[j][numb] = x[j][i];
  new_x_hier[j][numb] = x_hier[j][i];

  return 1;
}

/* Times Refinement */
static int times_refine(int j, int numb)
{
  int i;

  /*NAew Grid*/
  new_times[numb] = times[j - 1] + 0.5 * (times[j] - times[j - 1]);
  new_times[numb + 1] = times[j];
  new_times_hier[numb] = times_hier[j];
  new_times_hier[numb + 1] = times_hier[j];
  new_NA[numb] = NA[j];
  new_NA[numb + 1] = NA[j];

  for (i = 0; i <= new_NA[numb]; i++)
    {
      new_x[numb][i] = x[j][i];
      new_x[numb + 1][i] = x[j][i];
      new_x_hier[numb][i] = x_hier[j][i];
      new_x_hier[numb + 1][i] = x_hier[j][i];
    }

  return 1;
```

```
}

/* Times DeRefinement */
static int times_derefine(int j, int numb)
{
  int i;

  /*New Grid*/
  new_times[numb] = times[j + 1];
  new_times_hier[numb] = times_hier[j];
  new_NA[numb] = NA[j + 1];

  for (i = 0; i <= new_NA[numb]; i++)
    {
      new_x[numb][i] = x[j + 1][i];
      new_x_hier[numb][i] = x_hier[j + 1][i];
    }

  return 1;
}

/* Times OK */
static int times_ok(int j, int numb)
{
  int i;

  new_times[numb] = times[j];
  new_NA[numb] = NA[j];
  new_times_hier[numb] = times_hier[j];

  for (i = 0; i <= new_NA[numb]; i++)
    {
      new_x[numb][i] = x[j][i];
      new_x_hier[numb][i] = x_hier[j][i];
    }
  return 1;
}

static double g_func(int i, int j, double xi, double tj, double r, double divid)
{
  double a, b;
```

```
  a = (-divid * xi) * ((Primal_Price0[j][i] - Primal_Price0[j][i - 1]) / (x[j][i
  b = (1 + r * (tj - times[j - 1])) * (x[j][i - 1] * Primal_Price1[j][i] - x[j][
      + r * (x[j][i - 1] * Primal_Price0[j][i] - x[j][i] * Primal_Price0[j][i -
  return a + b;
}

/*Space error on finite elemente i,j*/
static double space_err(int i, int j, double r, double divid, double *j1s, doubl
{

  double w1, w3, rho1, rho3, j1, j2, j3, b, a, c, d;
  double norm_der_xx, norm_der_t_xx = 0.;

  double eta0s, eta1s, eta0t, eta1t, g0, g1, g2, g3;
  double p0, p1;

  /*Residual*/
  p0 = 0.211324865;
  p1 = 0.788675135;

  eta0s = p1 * x[j][i - 1] + p0 * x[j][i];
  eta1s = p0 * x[j][i - 1] + p1 * x[j][i];
  eta0t = p1 * times[j - 1] + p0 * times[j];
  eta1t = p0 * times[j - 1] + p1 * times[j];

  g0 = g_func(i, j, eta0s, eta0t, r, divid);
  g1 = g_func(i, j, eta0s, eta1t, r, divid);
  g2 = g_func(i, j, eta1s, eta0t, r, divid);
  g3 = g_func(i, j, eta1s, eta1t, r, divid);

  rho1 = sqrt(0.25 * (SQR(g0) + SQR(g1) + SQR(g2) + SQR(g3)) * (times[j] - times

  if (i == NA[j])
    norm_der_xx = sqrt((times[j] - times[j - 1]) * (x[j][i] - x[j][i - 1]) * ((4
  else if (i == 1)
    norm_der_xx = sqrt((times[j] - times[j - 1]) * (x[j][i] - x[j][i - 1]) * ((4
  else
    norm_der_xx = sqrt(0.5 * (times[j] - times[j - 1]) * (x[j][i] - x[j][i - 1])

  *NAORM2_XX += SQR(norm_der_xx) * (1. - 0.5 * (times[j] + times[j - 1]));
```

```
  w1 = norm_der_xx * SQR(x[j][i] - x[j][i - 1]);
  j1 = rho1 * w1;
  *j1s += j1;

  /*Jump of derivate*/
  j2 = 0.;
  *j2s += j2;

  /*Jump*/
  b = (x[j][i - 1] * (Primal_Proj[j][i] - Primal_Price0[j][i]) + x[j][i] * (Prim
  a = (Primal_Price0[j][i] - Primal_Price0[j][i - 1] - Primal_Proj[j][i] + Prima
  if (a == 0.)
    rho3 = (1. / sqrt(times[j] - times[j - 1])) * fabs(b) * sqrt(x[j][i] - x[j][
  else
    rho3 = (1. / sqrt(times[j] - times[j - 1])) * sqrt(fabs((CUB(a * x[j][i] + b

  if (i == NA[j])
    norm_der_xx = sqrt((times[j] - times[j - 1]) * (x[j][i] - x[j][i - 1]) * ((4
  else if (i == 1)
    norm_der_xx = sqrt((times[j] - times[j - 1]) * (x[j][i] - x[j][i - 1]) * ((4
  else
    norm_der_xx = sqrt(0.5 * (times[j] - times[j - 1]) * (x[j][i] - x[j][i - 1])

  if ((i != 1) && (i != NA[j]))
    {
      a = (Dual_Price1[j][i] - Dual_Proj1[j][i]) / (times[j] - times[j - 1]);
      b = (Dual_Price1[j][i - 1] - Dual_Proj1[j][i - 1]) / (times[j] - times[j -
      c = (Dual_Price1[j][i - 2] - Dual_Proj1[j][i - 2]) / (times[j] - times[j -
      d = (Dual_Price1[j][i + 1] - Dual_Proj1[j][i + 1]) / (times[j] - times[j -
      norm_der_t_xx = sqrt(0.5 * (times[j] - times[j - 1]) * (x[j][i] - x[j][i -
    }

  w3 = SQR(x[j][i] - x[j][i - 1]) * (norm_der_xx + norm_der_t_xx * (times[j] - t

  j3 = rho3 * w3;
  *j3s += j3;

  return j1 + j2 + j3;
}

/*Times error on finite elemente i,j*/
```

```
static double times_err(int i, int j, double r, double divid, double *j1t, doubl
{

  double w1, w3, rho1, rho3, j1, j2, j3, b, a, norm_der_tt;

  double eta0s, eta1s, eta0t, eta1t, g0, g1, g2, g3;
  double p0, p1;

  /*Residual*/
  p0 = 0.211324865;
  p1 = 0.788675135;

  eta0s = p1 * x[j][i - 1] + p0 * x[j][i];
  eta1s = p0 * x[j][i - 1] + p1 * x[j][i];
  eta0t = p1 * times[j - 1] + p0 * times[j];
  eta1t = p0 * times[j - 1] + p1 * times[j];

  g0 = g_func(i, j, eta0s, eta0t, r, divid);
  g1 = g_func(i, j, eta0s, eta1t, r, divid);
  g2 = g_func(i, j, eta1s, eta0t, r, divid);
  g3 = g_func(i, j, eta1s, eta1t, r, divid);

  rho1 = sqrt(0.25 * (SQR(g0) + SQR(g1) + SQR(g2) + SQR(g3)) * (times[j] - times

  if (j == M)
    {
      a = (2. / ((times[j] + times[j - 2]))) * (Dual_Price1[j][i] / (times[j] -
      b = (2. / ((times[j] + times[j - 2]))) * (Dual_Price1[j][i - 1] / (times[j
      norm_der_tt = sqrt(0.5 * (times[j] - times[j - 1]) * (x[j][i] - x[j][i - 1
    }
  else
    {
      a = (2. / ((times[j + 1] + times[j - 1]))) * (Dual_Price1[j][i] / (times[j
      b = (2. / ((times[j + 1] + times[j - 1]))) * (Dual_Price1[j][i - 1] / (tim
      norm_der_tt = sqrt(0.5 * (times[j] - times[j - 1]) * (x[j][i] - x[j][i - 1
    }

  w1 = norm_der_tt * SQR(times[j] - times[j - 1]);

  *NAORM2_T += SQR(norm_der_tt) * (1. - 0.5 * (times[j] + times[j - 1]));
  j1 = rho1 * w1;
```

```
  *j1t += j1;

  /*Jump of derivate*/
  j2 = 0.;
  *j2t += j2;

  /*Jump*/
  b = (x[j][i - 1] * (Primal_Proj[j][i] - Primal_Price0[j][i]) + x[j][i] * (Prim
  a = (Primal_Price0[j][i] - Primal_Price0[j][i - 1] - Primal_Proj[j][i] + Prima
  if (a == 0.)
    rho3 = (1. / sqrt(times[j] - times[j - 1])) * fabs(b) * sqrt(x[j][i] - x[j][
  else
    rho3 = (1. / sqrt(times[j] - times[j - 1])) * sqrt(fabs((CUB(a * x[j][i] + b
  w3 = w1;

  j3 = rho3 * w3;
  *j3t += j3;

  return j1 + j2 + j3;
}

/* Compute Error */
static double compute_global_error(int MAX_ADAPT, double r, double divid)
{
  int i, j, numb;
  double j1s, j2s, j3s, j1t, j2t, j3t, NAORM2_XX, NAORM2_T;

  n_element = 0;
  for (j = 1; j <= M; j++)
    n_element += NA[j];

  /*Space Error*/
  j1s = 0.;
  j2s = 0.;
  j3s = 0.;
  NAORM2_XX = 0.;
  global_space_error = 0.;
  for (j = 1; j <= M; j++)
    {
      for (i = 1; i <= NA[j]; i++)
        {
```

```
        space_error[j][i] = space_err(i, j, r, divid, &j1s, &j2s, &j3s, &NAORM
        global_space_error += space_error[j][i];
      }
  }

/*Times Error*/
global_times_error = 0.;
j1t = 0.;
j2t = 0.;

j3t = 0.;
NAORM2_T = 0.;
for (j = 1; j <= M; j++)
  times_error[j] = 0.;
for (j = 1; j <= M; j++)
  {
    for (i = 1; i <= NA[j]; i++)
      {
        times_error[j] += times_err(i, j, r, divid, &j1t, &j2t, &j3t, &NAORM2_
      }
    global_times_error += times_error[j];
  }

/*New Space Finite Element */
/*Refine*/
if (NA_ADAPT == MAX_ADAPT - 1) return global_space_error + global_times_error;
else
  {
    for (j = 1; j <= M; j++)
      new_x[j][0] = x[j][0];

    for (j = 1; j <= M; j++)
      {
        numb = 1;
        for (i = 1; i <= NA[j]; i++)
          {
            if ((space_error[j][i] > TOL_ADAPT_DUP / (2.*(double)n_element)) &
                (((((x_hier[j][i] == x_hier[j][i + 1]) && (x_hier[j][i] == x_h

                   ((x_hier[j][i] == x_hier[j][i + 1]) && (x_hier[j][i] == x_h
                   ((x_hier[j][i] == x_hier[j][i + 1] - 1) && (x_hier[j][i] ==
```

```
       ((x_hier[j][i] == x_hier[j][i + 1] - 1) && (x_hier[j][i] ==
  {
    x_hier[j][i]++;
    space_refine(j, i, numb);
    numb = numb + 2;
  }
else if ((space_error[j][i] > TOL_ADAPT_DUP / (2.*(double)n_elemen
         (((x_hier[j][i] == x_hier[j][i - 1])) ||
          ((x_hier[j][i] == x_hier[j][i - 1] - 1))))
  {
    x_hier[j][i]++;
    space_refine(j, i, numb);
    numb = numb + 2;
  }
/*Derefine*/
else if ((space_error[j][i] < (1. - EPS_MAX_ADAPT_DUP)*TOL_ADAPT_D
         (
           ((i != NA[j] - 1) &&
            (((x_hier[j][i] == x_hier[j][i + 2]) && (x_hier[j][i]
             ((x_hier[j][i] == x_hier[j][i + 2]) && (x_hier[j][i]
             ((x_hier[j][i] == x_hier[j][i + 2] + 1) && (x_hier[j]
             ((x_hier[j][i] == x_hier[j][i + 2] + 1) && (x_hier[j]
  {
    x_hier[j][i]--;
    x_hier[j][i + 1]--;
    space_derefine(j, i, numb);
    i++;
    numb++;
  }
else if ((space_error[j][i] < (1. - EPS_MAX_ADAPT_DUP)*TOL_ADAPT_D
         ((x_hier[j][i] == x_hier[j][i - 1]))))
  {
    x_hier[j][i]--;
    x_hier[j][i + 1]--;
    space_derefine(j, i, numb);
    i++;
    numb++;
  }
else
  {
    space_ok(j, i, numb);
```

```
            numb++;
          }
      }
    new_NA[j] = numb - 1;
  }

/*Save new space grid*/
for (j = 1; j <= M; j++)
  {
    NA[j] = new_NA[j];
    for (i = 0; i <= NA[j]; i++)
      {
        x[j][i] = new_x[j][i];
        x_hier[j][i] = new_x_hier[j][i];
      }
  }

/*New Times Finite Element */
/*Refine*/
times_hier[0] = times_hier[1];
numb = 1;
for (j = 1; j <= M; j++)
  {
    if ((times_error[j] > (1.*TOL_ADAPT_DUP) / (2.*(double)M)) && (j == M)
        (((times_hier[j] == times_hier[j - 1])) || ((times_hier[j] == time
      {
        times_hier[j]++;
        times_refine(j, numb);
        numb = numb + 2;
      }
    else if ((times_error[j] > (1.*TOL_ADAPT_DUP) / (2.*(double)M)) && (j
             (((times_hier[j] == times_hier[j + 1]) && (times_hier[j] == t
              ((times_hier[j] == times_hier[j + 1]) && (times_hier[j] == t
              ((times_hier[j] == times_hier[j + 1] - 1) && (times_hier[j]
              ((times_hier[j] == times_hier[j + 1] - 1) && (times_hier[j]
      {
        times_hier[j]++;
        times_refine(j, numb);
        numb = numb + 2;
      }
    else if ((sigma_type == 1) && ((times[j] == 0.25) || (times[j] == 0.5)
```

```
        {
          times_ok(j, numb);
          numb++;
        }
      else if ((times_error[j] < (1. - DELTA_MAX_ADAPT_DUP) * (1.*TOL_ADAPT_
               (((((times_hier[j] == times_hier[j + 2]) && (times_hier[j] ==
                  ((times_hier[j] == times_hier[j + 2]) && (times_hier[j] ==
                  ((times_hier[j] == times_hier[j + 2] + 1) && (times_hier[j
        {
          times_hier[j]--;
          times_hier[j + 1]--;
          times_derefine(j, numb);
          j++;
          numb++;
        }
      else if ((times_error[j] < (1. - DELTA_MAX_ADAPT_DUP) * (1.*TOL_ADAPT_
               && (times_hier[j] == times_hier[j + 1]) &&
               (((times_hier[j] == times_hier[j - 1])) ||
                ((times_hier[j] == times_hier[j - 1] + 1))))
        {
          times_hier[j]--;
          times_hier[j + 1]--;
          times_derefine(j, numb);
          j++;
          numb++;
        }
      else
        {
          times_ok(j, numb);
          numb++;
        }
    }
  new_M = numb - 1;

  /* Save new space and times grid */
  for (i = 0; i <= new_NA[1]; i++)
    {
      new_x[0][i] = new_x[1][i];
      new_NA[0] = new_NA[1];
    }
```

```
      new_times_hier[0] = new_times_hier[1];
      M = new_M;
      for (j = 0; j <= M; j++)
        {
          x_hier[j][0] = new_x_hier[j][1];
          times[j] = new_times[j];
          times_hier[j] = new_times_hier[j];
          NA[j] = new_NA[j];
          for (i = 0; i <= NA[j]; i++)
            {
              x[j][i] = new_x[j][i];
              x_hier[j][i] = new_x_hier[j][i];
            }
        }
    }
  times[0] = 0.;
  return global_space_error + global_times_error;
}

/* Grid Initialization */
static void init_grid_dupire()
{
  int i, j;



  for (j = 0; j <= M; j++)
    {
      for (i = 0; i <= NA[j]; i++)
        {
          x[j][i] = x_min + ((double)i) * (x_max - x_min) / (double)NA[j];
          x_hier[j][i] = 0;
        }
    }
}

/* Indicatrice Function */
static double theta_ind(double x, double a, double b)
{

  if ((x >= a) && (x <= b))
```

```
    return 1.;
  else return 0.;
}

/* Computation of error at maturity for the initializaton of dual problem */
static double theta_error()
{
  double error0, error1, eta0, eta1, price, price1, val, val1, bsnorm2;
  int i, NA_int, j1, j2;
  double *x_int;

  NA_int = 10000;
  bsnorm2 = 0.;
  x_int = malloc((NA_int + 1) * sizeof(double));

  for (i = 0; i <= NA_int; i++)
    x_int[i] = x_min + ((double)i) * (x_max - x_min) / (double)NA_int;

  for (i = 1; i <= NA_int; i++)
    {
      eta0 = 0.5 * (x_int[i] + x_int[i - 1]) - sqrt(3.) / 6.*(x_int[i] - x_int[i
      eta1 = 0.5 * (x_int[i] + x_int[i - 1]) + sqrt(3.) / 6.*(x_int[i] - x_int[i

      j1 = 1;
      while (x[M][j1] < eta0) j1++;

      j2 = 1;
      while (x[M][j2] < eta1) j2++;

      val = v_error[j1];
      val1 = v_error[j1 - 1];
      price = val + (val - val1) * (eta0 - x[M][j1]) / (x[M][j1] - x[M][j1 - 1])
      val = v_error[j2];
      val1 = v_error[j2 - 1];
      price1 = val + (val - val1) * (eta1 - x[M][j2]) / (x[M][j2] - x[M][j2 - 1]
      error0 = SQR(price) * theta_ind(eta0, a_ind, b_ind);
      error1 = SQR(price1) * theta_ind(eta1, a_ind, b_ind);
      bsnorm2 += (error0) * 0.5 * (x_int[i] - x_int[i - 1]) + (error1) * 0.5 * (
    }

  free(x_int);
```

```
  return sqrt(bsnorm2);
}

/* Initializaton of dual problem */
static void init_dual()
{
  int i, j;


  for (i = 1; i < NA_Coarse; i++)
    {
      j = 1;
      while (x[M][j] < x_Coarse[i]) j++;
      Proj[i] = Primal_Price[M][j - 1] * (x[M][j] - x_Coarse[i]) / (x[M][j] - x[
      v_error_Coarse[i] = Proj[i] - Price_Coarse[i];
    }

  v_error_Coarse[0] = 0.;
  v_error_Coarse[NA_Coarse] = 0.;

  v_error[0] = 0.;
  v_error[NA[M]] = 0.;

  for (i = 1; i < NA[M]; i++)
    {
      j = 1;
      while (x_Coarse[j] < x[M][i]) j++;
      v_error[i] = v_error_Coarse[j - 1] * (x_Coarse[j] - x[M][i]) / (x_Coarse[j
    }

  for (i = 0; i <= NA[M]; i++)
    {
      x_Coarse[i] = x[M][i];
      Price_Coarse[i] = Primal_Price[M][i];
    }
  NA_Coarse = NA[M];
  a_norm2 = theta_error();

  for (i = 0; i <= NA[M]; i++)
    v_error[i] = v_error[i] * theta_ind(x[M][i], a_ind, b_ind) / a_norm2;
```

```
}


/* Main Adaptive Procedure */
static int Adaptive(NumFunc_1  *p, double s, double t, double r, double divid, i
{

  int i, j;
  double global_error, val, val1, priceph, pricenh;
  double h = 0.00001;
  /* Memory Allocation */

  sigma_type = sigma;
  memory_allocation();

  x_min = s / 10.;
  x_max = s * 5.;
  NA_Coarse = NA0;
  for (i = 0; i <= NA0; i++)
    x_Coarse[i] = x_min + ((double)i) * (x_max - x_min) / (double)NA0;

  M_Coarse = M0;

  /*Problem on coarse mesh*/
  primal_solver0(p, s, t, r, divid, NA0, M0, Price_Coarse);

  /*Initial Times-Space Grid */
  for (j = 0; j <= M0; j++)
    {
      for (i = 0; i <= NA0; i++)
        x_Coarse1[j][i] = x_min + ((double)i) * (x_max - x_min) / (double)NA0;

      times_Coarse1[j] = (double)j * t / (double)M0;
      NA_Coarse1[j] = NA0;
    }

  NA0 = 2 * NA0;
  M = M0;

  for (j = 0; j <= M; j++)
    NA[j] = NA0;
```

```
init_grid_dupire();

for (j = 0; j <= M; j++)
  times[j] = (double)j * t / (double)M;

/* Adapt Cycle */
for (NA_ADAPT = 0; NA_ADAPT < MAX_ADAPT; NA_ADAPT++)
  {
    a_ind = s * 0.9;
    b_ind = s * 1.1;

    /*Solve the primal problem*/

    primal_solver(p, s, t, r, divid);

    if (NA_ADAPT == MAX_ADAPT - 1)
      {

        i = 0;
        while (x[M][i] < s)
          i++;

        val = Primal_Price[M][i];
        val1 = Primal_Price[M][i - 1];

        /*Price*/
        *ptprice = (val + (val - val1) * (s - x[M][i]) / (x[M][i] - x[M][i - 1

        /*Delta*/
        i = 0;
        while (x[M][i] < (s * (1 + h)))
          i++;
        val = Primal_Price[M][i];
        val1 = Primal_Price[M][i - 1];
        priceph = (val + (val - val1) * (s * (1. + h) - x[M][i]) / (x[M][i] -

        i = 0;
        while (x[M][i] < (s * (1. - h)))
          i++;
```

```
          val = Primal_Price[M][i];
          val1 = Primal_Price[M][i - 1];
          pricenh = (val + (val - val1) * (s * (1. - h) - x[M][i]) / (x[M][i] -

          *ptdelta = (priceph - pricenh) / (2.*s * h);

          free_memory();

          return OK;
        }

      /*Solve the dual problem*/
      init_dual();
      dual_solver(s, t, r, divid);

      /*Computation of indicator error*/
      global_error = compute_global_error(MAX_ADAPT, r, divid);

      /*Norm 2 at maturity*/
      *ptnorm2 = global_error;
    }
  return OK;
}

int CALC(FD_Adaptive)(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 Adaptive(ptOpt->PayOff.Val.V_NUMFUNC_1, ptMod->S0.Val.V_PDOUBLE,
                  ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE, r, divid, pt
}

static int CHK_OPT(FD_Adaptive)(void *Opt, void *Mod)
{
  /*
   * Option* ptOpt=(Option*)Opt;
```

```
   * TYPEOPT* opt=(TYPEOPT*)(ptOpt->TypeOpt);
   */
  if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt
    return OK;

  return WRONG;
}

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


      Met->Par[0].Val.V_INT2 = 32;
      Met->Par[1].Val.V_INT2 = 32;
      Met->Par[2].Val.V_INT2 = 5;


    }

  return OK;
}

PricingMethod MET(FD_Adaptive) =
{
  "FD_Adaptive",
  {{"First Space StepNumber", INT2, {100}, ALLOW}, {"First Time StepNumber", INT
  CALC(FD_Adaptive),
  {{"Price", DOUBLE, {100}, FORBID}, {"Delta", DOUBLE, {100}, FORBID}, {"Error I
  CHK_OPT(FD_Adaptive),
  CHK_ok,
  MET(Init),
};
```